

# **Programarea calculatoarelor**

**Îndrumar**

**C++**

<b>Lucrarea nr. 1 .....</b>	<b>7</b>
<b>Fundamentele limbajului C++. Partea I. ....</b>	<b>7</b>
Identificatori .....	7
Tipuri de date fundamentale .....	8
Constante .....	9
Variabile .....	13
Declarația typedef.....	18
Tipuri de date derivate.....	19
Tipuri de date definite de utilizator .....	23
<b>Operații de intrare / ieșire cu consola .....</b>	<b>31</b>
<b>Lucrarea nr. 2.....</b>	<b>37</b>
<b>Fundamentele limbajului C++. Partea II.....</b>	<b>37</b>
Operatori și expresii .....	37
Alocarea dinamică de memorie folosind operatorii new și delete .....	41
Operatorul de rezoluție (::) .....	43
Conversii de tip în expresii .....	44
Instrucțiuni.....	45
<b>Lucrarea nr. 3.....</b>	<b>52</b>
<b>Funcții în C/C++ .....</b>	<b>52</b>
Funcții.....	52
Definiții de funcții .....	52
Declarații de funcții. Prototipuri.....	53
Transferul parametrilor.....	54
Rezultatul unei funcții. Instrucțiunea return.....	56
Pointeri de funcții .....	57
Parametri cu valori implicite .....	59
Supradefinirea (supraîncărcarea) funcțiilor.....	61
Funcții inline.....	64
<b>Lucrarea nr. 4.....</b>	<b>68</b>
<b>Clase și obiecte (partea I).....</b>	<b>68</b>
Tipul class.....	69
Tipurile struct și union.....	72
Autoreferința. Cuvântul cheie “this” .....	73
Constructorii și destructorii .....	74
Crearea, inițializarea și eliminarea obiectelor .....	75
Constructor de copiere.....	77

<b>Lucrarea nr. 5.....</b>	<b>83</b>
<b>Clase și obiecte (partea II) .....</b>	<b>83</b>
Manevrarea dinamică a obiectelor.....	83
Transferul obiectelor ca parametri sau rezultat .....	84
Clase cu membri obiecte.....	87
Tablouri de obiecte .....	91
Pointeri către membrii unei clase. Operatorii ( .* ) și ( ->* ) .....	93
Membri statici ai unei clase .....	95
Funcții și clase prietene unei clase.....	97
<b>Lucrarea nr. 6.....</b>	<b>102</b>
<b>Supradefinirea operatorilor .....</b>	<b>102</b>
Funcția operator .....	102
Supradefinirea operatorilor folosind funcții membre clasei .....	103
Supradefinirea operatorilor folosind funcții prietene clasei .....	105
Supradefinirea operatorilor unari.....	107
Supradefinirea operatorului de atribuire (=).....	108
Supradefinirea operatorului [].....	114
Supradefinirea operatorilor new și delete .....	117
<b>Lucrarea nr. 7.....</b>	<b>121</b>
<b>Conversii de tip definite de utilizator .....</b>	<b>121</b>
Procedee de definire a conversiilor.....	121
Supradefinirea operatorului cast folosind funcții membre clasei .....	122
Supradefinirea operatorului cast folosind funcții prietene clasei .....	124
Conversii de tip folosind constructori.....	127
<b>Lucrarea nr. 8.....</b>	<b>132</b>
<b>Moștenirea. Clase derivate .....</b>	<b>132</b>
Declararea clasei derivate .....	132
Constructor și destructor pentru clasa derivată .....	136
Constructorul de copiere pentru o clasă derivată.....	138
Redefinirea funcțiilor membre.....	141
Compatibilitatea între o clasă derivată și clasa de bază. Conversii de tip .....	142
Supradefinirea operatorilor în clasele derivate .....	144
<b>Lucrarea nr. 9.....</b>	<b>148</b>
<b>Crearea unei ierarhii de clase.....</b>	<b>148</b>
<b>Moștenirea multiplă .....</b>	<b>151</b>
<b>Lucrarea nr. 10.....</b>	<b>158</b>
Clase virtuale .....	158
Funcții virtuale.....	162



## **Fundamentele limbajului C++. Partea I.**

### **Identificatori**

Identificatorii limbajului C++, ca și în limbajul C standard, sunt formați cu ajutorul caracterelor alfanumerice și liniuța de subliniere (**underscore**), “\_”. Primul caracter al unui identificator nu poate fi o cifră. De exemplu:

Raza	// valid
mesaj	// valid
_maxx	// valid
a5	// valid
5a	// invalid

Din mulțimea identificatorilor posibili, se remarcă **cuvintele cheie** (**reserved keywords**) ale limbajului, identificatori a căror semnificație nu poate fi modificată de programator (Tabelul nr. 1).

Cuvintele cheie din tabel care încep cu semnul **underscore** reprezintă variabile interne.

**Tabelul nr. 1.** Cuvinte cheie ale limbajului

Cuvinte cheie ale limbajului C++					
_asm	asm	auto	break	case	_cdecl
cdecl	char	class	const	continue	_cs
default	delete	do	double	_ds	else
enum	_es	_export	extern	_far	far
_fastcall	float	for	friend	goto	_huge
huge	if	inline	int	_interrupt	interrupt
_loadfs	long	_near	near	new	operator
_pascal	pascal	private	protected	public	register
return	_savefs	_seg	short	signed	sizeof
_ss	static	struct	switch	template	this
typedef	union	unsigned	virtual	void	volatile
while					

## Tipuri de date fundamentale

Tipul unei date determină dimensiunea zonei de memorie ocupate și valorile pe care le poate lua.

Tipurile datelor se pot grupa în **tipuri fundamentale** și **tipuri derivate**.

Tipurile de date fundamentale cuprind **tipurile aritmetice de bază** și **tipul void**.

Există patru tipuri aritmetice de bază, specificate prin cuvintele cheie: **char**, **int**, **float** și **double**. Gama de valori poate fi extinsă prin **modificatori de tip** desemnați prin cuvintele cheie: **signed**, **unsigned**, **short**, **long**. Tipurile întregi ce se obțin prin combinarea tipurilor de bază cu modificatorii de tip sunt prezentate în Tabelul nr. 2, iar cele reprezentate în virgulă mobilă în Tabelul nr. 3.

**Tabelul nr. 2.** Tipuri de date întregi

Tip	Spațiu de memorie ocupat	Domeniu de valori
char = signed char	8 biți	-128...127
unsigned char	8 biți	0...255
int = signed int = short int = signed short int	16 biți	-32768...32767
unsigned int = unsigned short int	16 biți	0...65535
long int = signed long int	32 biți	-2147483648...2147483647
unsigned long int	32 biți	0...4294967295

**Tabelul nr. 3.** Tipuri de date în virgulă mobilă

Tip	Spațiu de memorie ocupat	Domeniu de valori
float	32 biți	+/- (3.4E-38...3.4E+38) precizie 7 cifre
double	64 biți	+/- (1.7E-308...1.7E+308) precizie 15 cifre
long double	80 biți	+/- (3.4E-4932...1.1E4932) precizie 19 cifre

**Tipul fundamental void** indică absența oricărei valori și se utilizează în

următoarele situații: declarația unei funcții fără parametri sau care nu returnează un rezultat, tipul pointer generic și conversii de tip cu operatorul cast pentru pointeri.

Iată câteva exemple de expresii care returnează ca valoare spațiul de memorie ocupat de date de diferite tipuri:

sizeof (long int);	// expresia returnează valoarea 4
sizeof (unsigned char)	// expresia returnează valoarea 1
sizeof (long double)	// expresia returnează valoarea 10

**Observație:** C++ admite delimitatorii /\* \*/ pentru inserarea de comentarii care se pot întinde pe mai multe rânduri și, în plus, introduce delimitatorul // pentru includerea de comentarii de sfârșit de linie. Tot textul care urmează după delimitatorul // până la sfârșitul liniei este considerat comentariu.

## Constante

Constantele sunt valori fixe (numerice, caractere sau șiruri de caractere), care nu pot fi modificate în timpul execuției programului. Tipul constantei este determinat de compilator pe baza valorii și sintaxei utilizate. Ele rămân în memorie pe toată durata execuției programului.

### Constante întregi

Tipul constantei este determinat pe baza valorii, sau prin utilizarea unui sufix (U sau u pentru unsigned, respectiv L sau l pentru long).

Constantele întregi pot fi zecimale, octale sau hexazecimale.

### Constante zecimale (baza 10)

Constantele zecimale se disting prin faptul că prima cifră este diferită de 0, cum ar fi:

23	// tip int
23u	// tip unsigned int
32768	// tip long int
77UL	// tip unsigned long int

### Constante octale (baza 8)

Constantele octale sunt valori având prima cifră 0. Cifrele 8 și 9 sunt ilegale.

```
-067          // tip int
067u          // tip unsigned int
020000000    // tip long int
055ul        // tip unsigned long
089           // eroare, prima cifră indică reprezentarea în octal și numărul include
              // cifrele 8 și 9
```

### Constante hexazecimale (baza 16)

Constantele hexazecimale se disting prin prefixul 0x sau 0X. Pot conține cifre mai mari de 9 (a...f, sau A...F).

```
0x7FFF        // tip int
0X8000        // tip unsigned int
0xffu         // tip unsigned int
0x10000       // tip long int
0xFFul        // tip unsigned long int
```

### Constante în virgulă mobilă

Constantele în virgulă mobilă sunt valori raționale a căror reprezentare conține în general următoarele câmpuri:

- parte întreagă
- punct zecimal
- parte fracționară
- e sau E și un exponent cu semn (opțional)
- sufix de specificare a tipului: f sau F (forțază tipul float) sau l sau L (forțază tipul long double).

Se pot omite partea întreagă sau partea fracționară (dar nu amândouă), punctul zecimal sau litera e (E) și exponentul (dar nu amândouă).

Tipul implicit pentru constantele în virgulă mobilă este tipul **double**.

```
2.1           // valoare 2,1 (tip double)
11.22E5       // valoare 11,22 x 105 (tip double)
-.33e-2       // valoare -0,33 x 10-2 (tip double)
.5            // valoare 0,5 (tip double)
1.            // valoare 1 (tip double)
1.f           // valoare 1 (tip float)
0.L           // valoare 0 (tip long double)
```



## Constante caracter

Constantele caracter sunt reprezentate de unul sau mai multe caractere încadrate de apostrofuri, de exemplu: 'a', 'A', '\n' și ele sunt de tip char.

Pentru a specifica o serie de caractere neafișabile, delimitatorii ('), ("), caracterul (\), etc. se utilizează **secvențele escape** (secvențe de evitare) (vezi Tabelul nr. 4).

**Tabelul nr. 4.** Secvențe escape

Secvență	Caracter	Descriere
\a	alarm (bell)	sună alarma
\b	BS	backspace
\f	FF	formfeed
\n	LF	linefeed
\r	CR	carrige return
\t	TAB	tab orizontal
\v	VT	tab vertical
\\	\	backslash
\'	'	apostrof
\"	"	ghilimele
\?	?	semnul întrebării
\o	orice caracter	șir de cifre octale
\xH	orice caracter	șir de cifre hexazecimale

Exemple de utilizare a constantelor caracter:

```
#include <stdio.h>

void main()
{
    putchar('?');           // se afișează caracterul '?'
    putchar(63);           // se afișează caracterul '?', 63 fiind valoarea
                           // corespunzătoare în codul ASCII
    printf("\n%c", '\077'); // se afișează caracterul '?'
    printf("\n%c", '\x3F'); // se afișează caracterul '?'
}
```

**Observație:** În C, toate constantele de un singur caracter au tipul int și sunt reprezentate intern pe 16 biți cu octetul mai semnificativ 0 pentru valoarea caracterului mai mică decât 128, sau -1 (0xFF) pentru valori în intervalul 128...255.

## Constante caracter duble (specifice pentru C++)

Limbajul C++ permite specificarea unor constante alcătuite din două caractere, reprezentate pe 16 biți (tipul `int`) ca în exemplul care urmează:

```
| 'ab'           // reprezentarea în memorie ca întreg de valoare 25185 (0x62 0x61)
| '\t\t'        // reprezentarea în memorie ca întreg de valoare 2313 (0x09 0x09)
```

Primul caracter este memorat în octetul mai puțin semnificativ.

**Observație:** Constantele duble pot ridica probleme de portabilitate, ele nefiind recunoscute de alte compilatoare.

## Constante șiruri de caractere

Constantele șiruri de caractere sunt alcătuite dintr-un număr oarecare de caractere, încadrate între ghilimele.

Șirurile de caractere se memorează în tablouri de tipul `char`, cu dimensiune egală cu numărul de caractere cuprinse între ghilimele, la care se adaugă terminatorul de șir `'\0'`.

```
| "Exemplu de constanta sir de caractere"
```

Caracterele ce alcătuiesc șirul pot fi secvențe escape:

```
| "\tMesaj 1\n\tMesaj 2"
```

Șirurile constante adiacente se concatenează și formează un singur șir:

```
| "Programare" "orientata pe"
| "obiecte"
```

Pentru scrierea șirurilor lungi se poate utiliza simbolul (`\`) care semnalează continuarea șirului pe rândul următor:

```
| "Exemplu de sir \
| scris pe doua randuri"
```

## Variabile

### Declarații de variabile

Toate variabilele trebuie declarate înainte de a fi folosite.

Declarația unei variabile (obiect) precizează numele (identificatorul) cu care va fi referită, căruia îi poate asocia o serie de atribute, cum ar fi:

- **tipul** datei poate fi tip fundamental sau definit de utilizator și determină structura, gama de valori, dimensiunea spațiului ocupat în memorie;
- **clasa de memorare** stabilește zona de memorie în care se va plasa informația asociată identificatorului (segment de date, registru, stivă, heap) și delimitează durata sa de alocare;
- **domeniul** reprezintă porțiunea de program în care poate fi accesată informația asociată identificatorului, el fiind determinat de poziția declarației;
- **durata de viață** a identificatorului reprezintă perioada cât există efectiv în memorie și este corelată cu clasa de memorie;
- **legătura** precizează modul de asociere a unui identificator cu un anumit obiect sau funcție, în procesul de editare a legăturilor.

Atributele se pot asocia identificatorilor în mod implicit, în funcție de poziția și sintaxa declarației sau explicit prin utilizarea unor specificatori.

Sintaxa unei declarații de variabilă impune specificarea tipului, având forma generală:

**tip\_var nume\_var;**

tip\_var este un specificator de tip de date oarecare, standard, pointer sau definit de utilizator.

```
|| float * r;           // declararea variabilei r de tip pointer la float,  
|| unsigned int n;      // declararea variabilei n de tip unsigned int
```

Se pot face declarații de variabile cu inițializare utilizând sintaxa:

**tip\_var nume\_var= valoare\_iniciala;**

sau se pot declara mai multe variabile de același tip utilizând sintaxa:

**tip\_var nume\_var1<=val\_iniciala1>,< nume\_var2<= val\_iniciala2>>,...;**

```
|| double real=2.5;     // declararea variabilei real de tip double, inițializată cu valoarea 2.5
```

```
char c1, c2='a', ch;    // declararea a trei variabile de tip char, c1, c2 și ch, variabila c2
                        // fiind inițializată cu valoarea 'a'
```

Poziția declarației determină cele două domenii de existență fundamentale:

- **Domeniul bloc (local)**

Identificatorii cu domeniu bloc se numesc locali și sunt rezultatul unor declarații în interiorul unui bloc (au domeniul cuprins între declarație și sfârșitul blocului) sau sunt parametri formali din definiția unei funcții (au ca domeniu blocul funcției).

- **Domeniul fișier (global)**

Identificatorii cu domeniu fișier se numesc globali și sunt declarați în afara oricăror funcții (domeniul este cuprins între declarație și sfârșitul fișierului).

```
#include <stdio.h>

// zona declarațiilor globale

int functie (int, float);
int a;                // se declară variabila cu numele a, căreia i se rezervă o zonă de
                        // memorie de 2 octeți (16 biți) localizată în segmentul de date,
                        // deoarece declarația se află în zona declarațiilor globale

// definiții de funcții

void main (void)
{
    unsigned char c;    // se declară variabila automatică cu numele c căreia i se rezervă un
                        // octet în segmentul de stivă, variabila fiind o variabilă locală
    float r=2.5;        // se declară variabila automatică cu numele r căreia i se rezervă 4
                        // octeți în segmentul de stivă, variabila fiind o variabilă locală care este
    ...                // inițializată cu valoarea 2.5
}

int functie (int n, float q)    // se declară variabilele locale n și q, cărora li se alocă 2,
                                // respectiv 4 octeți în segmentul de stivă
{
    ...
}
```

**Observații:** 1. Într-un bloc inclus în domeniul unei declarații este permisă o declarație locală a aceluiași identificator, asocierea fiind însă făcută unui alt obiect cu alocarea altei zone de memorie.

2. Spre deosebire de C, care impune gruparea declarațiilor locale la începutul unui bloc, C++ permite plasarea declarațiilor în interiorul blocului, bineînțeles înainte de utilizarea obiectului.

```
void main (void)
{
...
for (int i=5; i<8; i++)      // declararea lui i este imediat urmată de utilizarea sa
{...}
for {i=0; i<5; i++})        // i a fost declarat anterior
{...}
...
}
```

Clasa de memorare se poate preciza prin specificatorii **auto**, **static**, **extern**, **register**. Acești specificatori precizează modul de alocare a memoriei și timpul de viață pentru variabile și legătură pentru funcții și variabile.

### **auto**

Declarația **auto** specifică o variabilă automatică și se poate utiliza pentru variabile cu domeniul local (cu spațiu alocat pe stivă). Variabilele ce se declară în interiorul unui bloc sunt implicit automate, astfel că declarația **auto** este rar folosită.

```
void fct()
{
    auto float r;           // declararea unei variabile automate cu declarare explicită
    double s;               // declararea unei variabile implicit automatică
    ...
}
```

### **static**

Declarația **static** a unei variabile locale forțează durată statică fără a modifica domeniul de existență. Variabilele statice își păstrează valoarea între două apeluri succesive ale blocurilor care le conțin, asigurându-se în același timp o protecție, dar ele nu pot să fie accesate din blocuri care nu constituie domeniul lor de existență. Variabilele statice pot fi declarate cu inițializare, în caz contrar, implicit se inițializează cu valoarea 0, similar variabilelor globale.

```
#include <stdio.h>

int fct()
{
    static int a=2;          // se declară o variabilă locală funcției, cu durată statică
    return (a++); }
void main(void)
{
    int n;
```

```
n=fct();
printf („\n Prima atribuire : n= %d”, n);
n=fct();
printf („\n A doua atribuire : n= %d”, n);
}
```

Programul afișează:

```
Prima atribuire : n= 3           // la primul apel al funcției, variabila a are valoarea
                                // inițială 2, ea fiind apoi incrementată
A doua atribuire : n= 4         // la al doilea apel al funcției, variabila a are la început
                                // valoarea 3 (valoare datorată apelului anterior al
                                // funcției), valoarea fiind apoi incrementată
```

### register

Declarația **register** are ca efect memorarea variabilei într-un registru al procesorului și nu în memorie, având ca rezultat creșterea vitezei de execuție a programului. Aceste variabilele pot fi variabile locale, nestatice, de tip `int` sau `char`. Numai două variabile pot fi memorate simultan în registre, în cazul existenței mai multor declarații `register`, cele care nu pot fi onorate vor fi tratate de compilator ca variabile obișnuite.

```
register char c;           // declararea variabilei c cu memorare într-un registru al procesorului
```

### extern

Specificatorul **extern** indică legătura externă și asigură durată statică pentru variabile locale și globale sau pentru funcții (acestea au implicit legătură externă și durată statică).

Pentru identificatorii cu legătură externă sunt permise mai multe declarații de referință, dar o singură definiție. De exemplu, în cazul unui proiect în care o variabilă se folosește în mai multe fișiere, ea se va defini global într-un fișier, în celelalte fiind declarată extern fără definire. Compilatorul îi va alocă o singură dată memorie.

```
// Fișier Ex_prj.cpp

#include <stdio.h>

extern int m;           // se declară legătură externă pentru variabila m

void main(void)
{
    ...
    scanf(“%d”, &m);
    printf(“\nm= %#x”, m);
}
```

```
...
}

// Fișier Ex_prj1.cpp

...
int m ;                // declarația variabilei m
...
```

În exemplul anterior, cele două fișiere, Ex\_prj.cpp și Ex\_prj1.cpp, se includ într-un proiect utilizând opțiunea “project” a meniului principal din mediul de programare C/C++. Variabila m este declarată global într-unul dintre fișiere, ea putând fi referită din celălalt fișier datorită specificării legăturii “extern” .

## Modificatori de acces

Modificatorii de acces ce pot fi utilizați sunt **const** și **volatile** și ei pot controla modul de modificare a unei variabile.

### const

Variabilele **const** nu pot fi modificate pe parcursul execuției unui program. Instrucțiunile care încearcă modificarea variabilelor const generează erori la compilare.

```
const int a=99;        // declarația variabilei a const
a++;                  // eroare, a este const
a+=5;                 // eroare, a este const
```

O variabilă pointer constantă nu se poate modifica, în schimb se poate modifica obiectul indicat.

În cazul unui pointer către un obiect constant, valoarea pointerului este modificabilă, dar nu și valoarea obiectului.

```
void main()
{
    int var1=25, var2=44;
    const int var3=7;
    int * p1=&var1;        // declarație cu inițializare
    int * p2=&var3;        // eroare, nu se poate converti 'const int *' la 'int *'
    const int * p3=&var3;  // corect, tipul de date corespund
    var3++;               // eroare, se încearcă modificarea unui obiect constant
}
```

```
*p3=15;           // eroare, se încearcă modificarea unui obiect constant
int * const p4=&var1; // corect, se declară un pointer constant
p4=&var2;          // eroare, se încearcă modificarea unui pointer constant
*p4++;            // corect, se modifică valoarea obiectului var2 care nu
                  // este declarat constant
}
```

În mod uzual se folosesc funcții cu parametri pointeri către obiecte constante atunci când se dorește protejarea la modificare a acestora.

Un exemplu este o funcție care realizează criptarea unui șir de caractere:

```
#include <stdio.h>
#include <string.h>

void criptare(const char *sir_i, char *sir_c); // parametrul sir_i este declarat const pentru a
                                              // evita modificarea lui în interiorul funcției
                                              // criptare()

void main()
{
    char sir_initial[80], sir_criptat[80];
    sir_initial="Exemplu criptare";
    criptare(sir_initial, sir_criptat);
    printf("\nSirul criptat este: %s", sir_criptat);
}

void criptare( const char *sir_i, char *sir_c)
{
    while (*sir_i)
    {
        *sir_c = * sir_i+1 ;           // criptarea se face prin modificarea codului ASCII
        sir_i ++ ;
        sir_c ++ ;
    }
    *sir_c='\\0';
}
```

## volatile

Variabilele **volatile** pot fi modificate din exteriorul programului (de exemplu servirea unei întreruperi).

## Declarația typedef

Specificatorul **typedef** nu declară un obiect, ci asociază un nume unui tip



de date. Sintaxa este:

### **typedef tip\_data identificator\_tip**

```
typedef unsigned char octet;      // tipul unsigned char este echivalent cu octet
octet var;                       // variabila var se declară de tip octet
```

## **Tipuri de date derivate**

### **Pointeri de date**

Tipul **pointer** (indicator) reprezintă adrese ale unor zone de memorie (adrese de variabile sau constante).

Există, de asemenea, pointeri generici, numiți pointeri void, care pot conține adresa unui obiect de orice tip.

### **Declararea variabilelor pointer**

Sintaxa declarației unui pointer de date este:

**tip \* id\_ptr;**

Simbolul \* precizează că id\_ptr este numele unei variabile pointer, iar tip este tipul obiectelor a căror adresă o va conține (tipul de bază al id\_ptr).

Compilerul interpretează zona de memorie adresată de pointer ca obiect de tipul indicat în declarație, cu toate atributele tipului: dimensiunea zonei de memorie necesară și semnificația informației conținute.

Declararea unui pointer generic se face folosind sintaxa:

**void \* id\_vptr;**

În cazul acestei declarații, dimensiunea zonei de memorie adresate și interpretarea informației nu sunt definite, iar proprietățile diferă de ale celorlalți pointeri de obiecte.

```
int * ptr;           // pointer la întreg
int * tabptr[10];    // tablou de pointeri către întregi
float **pptr;        // dublă indirectare; pointer la pointer
```

Este important că orice variabilă pointer trebuie inițializată cu o valoare

validă, 0 (NULL) sau adresa unui obiect înainte de a fi utilizată. În caz contrar, efectele pot fi grave deoarece la compilare sau în timpul execuției nu se fac verificări ale validității valorilor pointerilor.

### ***Operatori specifici.***

Există doi operatori unari care permit folosirea variabilelor pointer: operatorul **&** se folosește pentru aflarea adresei unei variabile oarecare și operatorul **\*** pentru accesul la variabila adresată de un pointer.

```
#include <stdio.h>

void main(void)
{
    int var=20, *pvar;
    printf("\nVariabila var se afla la adresa %p", &var);
    printf("si are valoarea var= %d", var);
    pvar=&var;
    printf("\nVariabila pvar are valoarea %p", pvar);
    printf("si adreseaza obiectul: %d", *iptr);
    *iptr=50;
    printf("\nNoua valoare a lui var este %d", var);
}
```

Programul afișează:

```
Variabila var se află la adresa: FFF4 si are valoarea iv= 20
Variabila pvar are valoarea: FFF4 si adreseaza obiectul: 20
Noua valoare a lui var este 50
```

Situația:

```
tip1 * id_pl;
tip2 * id_p2;
id_pl=&id_p2;
```

nu generează erori dacă tip1 și tip2 sunt identice, sau, în caz contrar dacă tip1 este void.

Dacă se folosește un pointer void, pentru orice referire a obiectului adresat este necesară precizarea explicită a tipului utilizând operatorul cast.

```
void main()
{ void *p1;
  float *p2, var=1.5;
  p2=&var;
  p1=p2;
```

```
printf("\nValoarea referita de p1: %f", *(float *) p1);  
}
```

## Variabile referință

C++ oferă posibilitatea de a declara identificatori ca referințe de obiecte (variabile sau constante). Referințele, ca și pointerii, conțin adrese. Pentru a declara o referință la un obiect se folosește simbolul &, folosind sintaxa:

**tip & id\_referinta = nume\_obiect;**

tip este tipul obiectului pentru care se declară referința, iar simbolul & precizează că id\_referinta este numele unei variabile referință, iar nume\_obiect este obiectul a cărui adresă va fi conținută în id\_referinta.

```
int n;                // se declară n de tip întreg  
int * p=&n;           // se declară pointerul p cu inițializare adresa lui n  
int &r=n;              // se definește r referința lui n  
  
n=20;                 // n primește valoarea 20  
*p=25;                // n primește valoarea 25  
r=30;                 // n primește valoarea 30
```

În exemplul anterior, atât p, cât și r, acționează asupra variabilei n.

Atunci când se accesează o variabilă prin referința sa, nu este necesar să se folosească adresa, acest lucru realizându-se automat.

Spre deosebire de pointeri, care la un moment dat pot primi ca valoare adresa unei alte variabile, referințele nu pot fi modificate, ele fiind practic o redenumire a variabilei a căror adresă o conțin (se creează un alias al respectivei variabile).

În utilizarea referințelor, trebuie avute în vedere următoarele restricții:

- referințele trebuie inițializate în momentul declarării;
- referințelor nu li se pot modifica locațiile la care se referă;
- nu sunt permise referințe la câmpuri de biți, referințe la referințe și pointeri la referințe, deci nici tablouri de referințe.

```
int &r ;                // eroare, se declară o referință fără inițializare
```

```
int &r=20 ;             // corect, se declară o referință la o constantă
```

```
const int i = 20;  
int &r = i ;            // eroare, se declară o referință întreagă la o constantă întreagă
```

```
const int i = 20;  
const int &r = i ;           // corect, se declară o referință constantă întreagă la o constantă  
                             // întreagă
```

**Observație:** Referințele de sine stătătoare sunt rar folosite. În schimb, utilizarea parametrilor formali referință permite transferul prin referință simplu și eficient, fără recurgerea la parametri formali pointeri.

## Tablouri de date

Tabloul de date (sau masiv de date) este o colecție de date de același tip, plasate într-o zonă contiguă de memorie.

Sintaxa declarației unui tablou cu N dimensiuni este:

**tip\_element nume\_tablou [dimensiune1][dimensiune2]...[dimensiuneN]**

Zona de memorie rezervată conține

dimensiune1 x dimensiune2 x...x dimensiuneN

elemente de tipul tip\_element.

Referirea unui element de tablou se face cu operatorul de indexare [] sub forma:

**nume\_tablou [indice1][indice2]...[indiceN]**

Declarația unui tablou se poate face cu inițializarea sa folosind sintaxa:

**declaratie\_tablou={listă\_valori};**

Lista de valori trebuie să conțină constante de tip compatibil cu tipul de bază al tabloului, în ordinea plasării în memorie.

```
float vect1[5];           // se declară un tablou cu 5 elemente float, fără inițializare  
vect1[0]=1.5;             // elementului de index 0 i se atribuie valoarea 1.5  
int vect2[10]={2,7,-1,0,9,15,-5,22,6,11}; // se declară un tablou cu 10 elemente int cu  
                           // inițializare  
int mat[2][3]={ {3,5,-3}, {2,-1,0} }; // se declară un tablou bidimensional cu 2*3  
                           // elemente de tip întreg, cu inițializare  
mat[1][2]= 23;           // elementului de indecși 1, respectiv 2, i se  
                           // atribuie valoarea 23
```

Tablourile unidimensionale cu elemente de tip char sunt folosite pentru memorarea șirurilor de caractere. Pentru a marca sfârșitul unui șir de caractere, după ultimul caracter se adaugă un octet cu valoarea 0 ('\0'), numit și terminator

de șir.

```
char sir1[10];           // se declară un tablou unidimensional, cu elemente char
                        // (șir de caractere), fără inițializare
char sir3[ ]="sir de caractere"; // se declară un tablou cu 17 elemente char (șir de
                        // caractere) cu inițializare (ultimul caracter depus în
                        // tablou este terminatorul de șir '\0'
sir3[0]='S';             // primul caracter al șirului primește valoarea 'S'
```

Numele unui tablou fără index este un pointer constant de tipul elementelor tabloului și are ca valoare adresa primului element al tabloului.

```
float tab[20], *ptr;
ptr=tab;                // atribuire validă, pointerul ptr va conține adresa
                        // primului element al tabloului

&tab[0] == tab ;
&tab[2] == tab+2 ;
tab[0] == *tab ;
tab[2] == *(tab+2) ;
tab++;                 // eroare, tab este un pointer constant, deci nu se poate incrementa
ptr++;                 // corect, ptr este un pointer la float, nu a fost declarat constant
```

Tablourile multidimensionale reprezintă tablouri cu elemente tablouri, astfel încât numele tabloului (fără index) este un pointer de tablouri.

```
float mat[10][10];      // mat reprezintă un tablou de pointeri float
float *p;               // p este un pointer float
p=mat;                  // eroare, tipurile pointerilor diferă
p=(float*)mat;          // corect, s-a folosit o conversie explicită de tip
mat == &mat[0][0];      // expresie adevărată
mat+1 == &mat[1][0];    // expresie adevărată
*(mat+9) == mat[9][0];  // expresie adevărată
```

## Tipuri de date definite de utilizator

Tipurile de date fundamentale și derivate nu pot acoperi totdeauna necesitățile de organizare a informației, în numeroase situații fiind necesară asocierea de date de tipuri diferite.

Vom prezenta în continuare posibilitățile oferite de limbajul C pentru definirea unor tipuri de date cu un grad de complexitate sporit. Acestea sunt:

- enumerarea, care este o listă de identificatori cu valori constante de tip întreg;
- structura, care este o colecție de date de tipuri diferite referite cu același nume;
- câmpul de biți, care este un membru al unei structuri căruia i se alocă în

memorie un număr de biți în interiorul unui cuvânt;

- uniunea, care permite utilizarea aceleiași zone de memorie de obiecte de tipuri diferite.

Noțiunile puse în discuție în această lucrare de laborator sunt cele comune celor două limbaje, urmând ca, într-o lucrare ulterioară tratarea acestora să se completeze cu aspecte specifice limbajului C++.

## Enumerarea

Tipul enumerare constă dintr-un ansamblu de constante întregi, fiecare asociată unui identificator. Sintaxa declarației este:

**enum <id\_tip\_enum> {id\_elem<=const>,...} <lista\_id\_var>;**

unde: id\_tip\_enum = nume tip enumerare;  
id\_elem = nume element;  
const = constantă de inițializare a elementului.

Dacă declarația nu specifică o constantă, valorile implice sunt 0 pentru primul element, iar pentru celelalte valoarea elementului precedent incrementat cu o unitate.

Identificatorii elementelor trebuie să fie unici în domeniul lor (diferiți de numele oricărei variabile, funcție sau tip declarat cu typedef).

```
| enum boolean {false, true}; // valoarea identificatorului false este 0, iar a lui true este 1  
| sau  
| typedef enum {false, true} boolean; // declarația este echivalentă declarației anterioare
```

```
| # include <stdio.h>  
  
| enum boolean {false, true}; // valoarea identificatorului false este 0, iar a lui true este 1  
  
| void main()  
| {  
|     boolean op1=false, op2;  
|     op2=true;  
|     printf("\n op1=%d\n op2=%d", op1, op2);  
| }
```

Tipul enumerare facilitează operarea cu variabile care pot lua un număr mic de valori întregi, asociindu-se nume sugestive pentru fiecare valoare. Programul devine mai clar și mai ușor de urmărit.

## Structuri

Structura este o colecție de date referite cu un nume comun. O declarație de structură precizează identificatorii și tipurile elementelor componente și constituie o definiție a unui nou tip de date.

Sintaxa declarației unui tip structură este:

```
struct id_tip_struct {  
    tip_elem1 id_elem1;  
    tip_elem2 id_elem2;  
    ...  
    tip_elemN id_elemN;} lista_id_var_struct;
```

unde: struct = cuvânt cheie pentru declararea tipurilor structură;

id\_tip\_struct = numele tipului structură declarat;

tip\_elemK = tipul elementului K, unde K=1...N;

id\_elemK = numele elementului K;

lista\_id\_var\_struct = lista cu numele variabilelor de tipul declarat, id\_ti\_struct.

Pot să lipsească, fie numele structurii, fie lista variabilelor declarate, dar nu amândouă. De regulă se specifică numele tipului structură definit, aceasta permițând declarații ulterioare de obiecte din acest tip.

```
struct data                // se declară tipul de date data, ca structură cu 3 membri de tip  
                           // unsigned int  
{ unsigned int zi;  
  unsigned int luna;  
  unsigned int an;  
};  
struct persoana            // se declară tipul de date persoana ca structură cu 2  
                           // membri șir de caractere și un membru de tip data  
{  
  char nume[15];  
  char prenume[15];  
  data data_n;  
} pers1, pers2;           // odată cu declarația tipului de date persoana, se declară  
                           // două obiecte din acest tip de date
```

Se poate declara un tip structură cu ajutorul declarației typedef cu sintaxa:

```
typedef struct {  
    tip_elem1 id_elem1;  
    tip_elem2 id_elem2;
```

```
...  
tip_elemN id_elemN;} id_tip_struct;
```

unde semnificația denumirilor este identică cu cea anterioară.

Se reia exemplul anterior, utilizând typedef pentru declarația structurilor:

```
typedef struct  
{  
    unsigned int zi;  
    unsigned int luna;  
    unsigned int an;  
} data;  
  
typedef struct  
{  
    char nume[15];  
    char prenume[15];  
    data data_n;  
} persoana;
```

Elementele structurii se numesc generic membrii (câmpurile) structurii. Tipurile membrilor pot fi oarecare, mai puțin tipul structură care se definește, dar pot fi de tip pointer la structura respectivă.

Inițializarea unei variabile structură se face prin enumerarea valorilor membrilor în ordinea în care apar în declarație.

```
persoana pers={"Ionescu", "Adrian", 10, 10, 1975};    // declararea unui obiect persoana  
                                                    // cu inițializare
```

Referirea unui membru al unei variabile de tip structură se face folosind operatorul de selecție (.) (punct), sub forma:

**nume\_variabilă . nume\_câmp**

```
printf("\nNumele: %s", pers.nume);  
printf("\nPrenume: %s", pers.prenume);  
printf("\nData nasterii: %d.%d.%d", pers.data_n.zi, pers.data_n.luna, pers.data_n.an);
```

Referirea unui membru al unei structuri indicate de un pointer se face folosind operatorul de selecție indirectă (->) (săgeată).

```
persoana * p_pers;                // se declară un obiect pointer la persoana  
  
puts("\nIntroduceti numele:");
```



```
scanf(„%s”, &p_pers->nume);
puts(„\nIntroduceti prenumele:”)
scanf(„%s”, &p_pers->prenume);
puts(„\nIntroduceti data nasterii:”);
scanf(„%d.%d.%d”, &p_pers->data_n.zi, &p_pers->data_n.luna, &p_pers->data_n.an);
```

Operatorul de atribuire admite ca operanzi variabile structură de același tip, efectuând toate atribuirile membru cu membru.

```
persoana p1={„Popescu”, „George”, 5, 12, 1982}, p2;    // p1 se declară cu inițializare
p2=p1;                                                  // prin atribuire, p2 preia, membru
                                                        // cu membru datele din p1

printf(„\nNumele:%s”, p2.nume);
printf(„\nPrenume:%s”, p2.prenume);
printf(„\nData nasterii: %d.%d.%d”, p2.data_n.zi, p2.data_n.luna, p2.data_n.an);
```

Transferul unui parametru de tip structură se poate face prin valoare. Parametrul formal și cel efectiv trebuie să fie de același tip.

```
#include <stdio.h>

typedef struct complex {int re, im;} ;

complex suma(complex a, complex b)
{
    complex c;
    c.re=a.re+b.re;
    c.im=a.im+b.im;
    return c;
}

void main()
{
    complex c1, c2, c3;
    scanf(„%d”, &c1.re);
    scanf(„%d”, &c1.im);
    scanf(„%d”, &c2.re);
    scanf(„%d”, &c2.im);
    c3=suma(c1, c2);
    printf(„\n%i %i”, c3.re, c3.im);
}
```

Transferul unui parametru de tip structură se poate face prin referință. Parametrii efectivi trebuie să fie adrese de structuri de același tip cu parametrii.

```
#include <stdio.h>
```

```
typedef struct complex {int re, im;} ;

complex suma(complex *a, complex *b)
{
    complex c;
    c.re=a->re+b->re;
    c.im=a->im+b->im;
    return c;
}

void main()
{
    complex c1, c2, c3;
    c1.re=12;
    c1.im=-7;
    c2.re=29;
    c2.im=35;
    c3=suma(&c1, &c2);
    printf("\n%i  %i", c3.re, c3.im);
}
```

Pentru transferul parametrilor de tip structură se pot folosi variabile referință de structură de același tip.

```
#include <stdio.h>
typedef struct complex {int re, im;} ;
complex suma(complex &a, complex &b)
{
    complex c;
    c.re=a.re+b.re;
    c.im=a.im+b.im;
    return c;
}

void main()
{
    complex c1, c2, c3;
    puts("\nIntroduceti valorile:")
    scanf("%d", &c1.re);
    scanf("%d", &c1.im);
    scanf("%d", &c2.re);
    scanf("%d", &c2.im);
    c3 = suma(c1,c2);
    printf("\nc3=%d + i* %d", c3.re, c3.im);
}
```



```
data_n.zi=aux1;
data_n.luna=aux2;
data_n.an=aux3;
....}
```

Se poate observa că zona de memorie ocupată de un obiect date este de 3 octeți, spre deosebire de structura care nu include câmpuri de biți care ocupă 6 octeți.

```
| printf("data_n ocupa %d octeti", sizeof(data_n));
```

## Uniuni

Uniunea permite utilizarea în comun a unei zone de memorie de către mai multe obiecte de tipuri diferite. Sintaxa de declarare a unei uniuni este similară declarației unei structuri:

```
union id_tip_uniune {
    tip_elem1 id_elem1;
    tip_elem2 id_elem2;
    ...
    tip_elemN id_elemN;} lista_id_var_uniune;
```

sau

```
typedef union {
    tip_elem1 id_elem1;
    tip_elem2 id_elem2;
    ...
    tip_elemN id_elemN;} id_tip_uniune;
```

Spațiul alocat în memorie corespunde tipului de dimensiune maximă, membrii uniunii utilizând în comun zona de memorie.

```
#include <stdio.h>

struct octet{    unsigned int b0:1;    // se declară o structură care ocupă un octet de memorie,
                unsigned int b1:1;    // cu acces separat, prin membrii săi, la fiecare bit în
                unsigned int b2:1;    // parte
                unsigned int b3:1;
                unsigned int b4:1;
                unsigned int b5:1;
                unsigned int b6:1;
                unsigned int b7:1;

                };
```

```
union intreg
{
    char val;           // se declară o uniune ce ocupă un octet de memorie care poate
    octet bit;          // fi accesat ca și char prin membrul val, sau ca octet prin
                        // membrul bit
};

void main ()
{
    intreg i;
    i.val=22;
    // se afișează fiecare bit al uniunii separat:
    printf("\n0x%x se reprezintă în binar: %d%d%d%d%d%d%d", i.val, i.bit.b7,
        i.bit.b6, i.bit.b5, i.bit.b4, i.bit.b3, i.bit.b2, i.bit.b1, i.bit.b0);
}
```

Programul afișează:

|| 0x16 se reprezintă în binar 00010110

## Operații de intrare / ieșire cu consola

C++ permite utilizarea funcțiilor de intrare/ieșire C, însă dispune de un sistem de intrare/ieșire conceput în spiritul POO, mai flexibil și mai comod. Acest sistem este prezentat sumar în continuare.

În C++ sunt predefinite dispozitive logice de intrare/ieșire standard similare celor din limbajul C:

- **cin** = console input = dispozitiv de intrare consolă, tastatura (echivalent cu stdin din C);
- **cout** = console output = dispozitiv de ieșire consolă, monitorul (echivalent cu stdout din C).
- **cerr** = dispozitiv de ieșire pentru afișarea erorilor, fără memorare (echivalent cu stderr din C).
- **clog** = dispozitiv de ieșire pentru afișarea erorilor, cu memorare (nu are echivalent în C).

Transferul informației cu formatare este efectuat de operatorul >> pentru intrare (de la cin) și de << pentru ieșire (către cout), deci:

```
cin >> var;           /* citește var de la cin */
```

**cout << var;**      /\* scrie var la cout \*/

Sunt posibile operații multiple, de tipul:

**cin >> var1 >> var2 ... >> varN;**  
**cout << var1 << var2 ... << varN;**

În acest caz, se efectuează succesiv, de la stânga la dreapta, scrierea la cout, respectiv citirea de la cin a valorilor var1 ... varN.

Tipurile datelor transferate către cout pot fi:

- toate tipurile aritmetice;
- șiruri de caractere;
- pointeri de orice tip în afară de char.

Tipurile datelor citite de la cin pot fi:

- toate tipurile aritmetice;
- șiruri de caractere.

Controlul formatului pentru ambele operații este posibil, dar nu este obligatoriu deoarece există formate standard. Acestea sunt satisfăcătoare pentru dialogul cu consola efectuat în exemplele din capitolele următoare.

Pentru citirea/scrierea șirurilor de caractere se poate specifica o constantă șir sau un pointer de caractere. Din acest motiv, pentru afișarea adresei unui șir este necesară o conversie explicită la pointer de alt tip, de exemplu (void \*). Valorile adreselor se afișează în hexazecimal.

Operațiile de citire de la tastatură efectuate cu operatorul >> sunt similare cu cele efectuate cu scanf(). Delimitatorii câmpurilor introduse sunt spațiu, tab, linie nouă, etc. În cazul citirii unui caracter nevalid, citirea este întreruptă și caracterul rămâne în tamponul de intrare generând probleme similare cu cele care apar la utilizarea funcției scanf().

Utilizarea dispozitivelor și operatorilor de intrare/ieșire C++ impune includerea fișierului antet **iostream.h**. Exemplul următor este elocvent pentru eleganța și simplitatea dialogului cu consola în C++.

```
#include <iostream.h>
void main()
{
    int i;
    char nume[21];
    float r;
    cout << "introduceti un numar intreg si apoi un numar real: " ;
    cin>> i >> r;
    cout << "\nAti introdus: "<< i << "si" << r << "\n";
}
```

```
cout << "Introduceti numele dvs: ";
cin >> nume;
cout<< "Salut, " << nume << " !\n" ;
}
```

Programul afișează:

```
Intruduceti un numar intreg si apoi un numar real:15 3.1416
Ati introdus:15 si 3.1416
Introduceti numele dvs: ANDREI
Salut, ANDREI !
```

Pentru afișare se pot utiliza expresii:

```
#include <iostream.h>

int main() {
    int num;
    cin >> num;
    cout << num + 1;    } // se afișează rezultatul returnat de expresia "num+1"
```

Pentru citirea/scrierea șirurilor de caractere se poate specifica o constantă șir sau un pointer de caractere. Din acest motiv, pentru afișarea adresei unui șir este necesară o conversie explicită la pointer de alt tip, de exemplu (void \*). Valorile adreselor se afișează în hexazecimal.

```
#include <iostream.h>

void main()
{
    char sir[20]="Sir de caractere"; // se declară un șir de caractere cu inițializare
    cout<<sir<<"\n"; // se afișează conținutul șirului de caractere "sir"
    cout<<*sir<<"\n"; // se afișează primul caracter din șirul de caractere "sir"
    cout<<&sir<<"\n"; // se afișează adresa la care se află variabila pointer "sir"
    cout<<(void*)sir<<"\n"; // se afișează adresa la care se află variabila pointer "sir"
    cin>>*sir; // se citește un alt caracter pentru prima poziție a șirului
    cout<<sir<<"\n"; // se afișează conținutul șirului de caractere "sir"
    cin>>sir; // se citește o nouă valoare pentru șirul de caractere
    cout<<sir<<"\n"; // se afișează conținutul șirului de caractere "sir"
    char * p_sir="abc"; // declară un pointer la tipul char ce se inițializează cu
    // adresa șirului constant "abc"
    cout<<p_sir<<"\n"; // se afișează conținutul șirului de caractere către care
    // pointează p_sir
    cout<<*p_sir<<"\n"; // se afișează primul caracter din șirul constant de
    // caractere
    cout<<&p_sir<<"\n"; // se afișează adresa la care se află variabila pointer
    // p_sir
    cout<<(void*)p_sir<<"\n"; // se afișează adresa conținută de variabila pointer p_sir,
```

```
| // deci adresa la care se află șirul constant "abc"  
| }
```

Programul afișează:

```
| Sir de caractere  
| S  
| 0xffe0  
| 0xffe0  
| s //caracterul introdus de la tastatură  
| sir de caractere  
| Alt sir de caractere // șirul introdus de la tastatură  
| Alt sir de caractere  
| abc  
| a  
| 0xfff4  
| 0x00be
```

Concatenarea șirurilor de caractere la afișare se poate realiza ca în exemplul următor:

```
| #include <iostream.h>  
  
| int main()  
| {  
|     cout << "Acesta este un sir prea lung "  
|         "pentru a fi scris pe un singur rand.\n"  
|         "El poate fi continuat pe randul urmator.\n"  
| }  
| }
```

Programul afișează:

```
| Acesta este un sir prea lung pentru a fi scris pe un singur rand.  
| El poate fi continuat pe randul urmator.
```

**Observație:** Trebuie reținut că operatorii `>>` și `<<` își păstrează și semnificația din C - operatori de deplasare bit cu bit la dreapta, respectiv la stânga.

Pentru formatarea intrărilor și ieșirilor sunt definiți o serie de manipulatori. De exemplu, întregii pot fi însoțiți de **dec**, **oct**, **hex**, pentru reprezentarea în zecimal, octal, respectiv hexazecimal. Valoarea implicită este dec. De asemenea, se poate folosi **endl** care inserează caracterul `'\n'` și golește



tamponul de ieșire și **ends** care inserează caracterul ‘\0’ (terminator de șir de caractere).

// Specificarea formatului de afișare cu utilizarea manipulatorilor

```
#include <iostream.h>
```

```
int main() {  
    cout << "un numar in zecimal: " << dec << 15 << endl;  
    cout << "in octal: " << oct << 15 << endl;  
    cout << "in hex: " << hex << 15 << endl;  
    cout << "un numar in virgula mobila: " << 3.14159 << endl;  
    cout << "caracter neafisabil (escape): " << char(27) << endl;  
}
```

// Convertirea unui numar în zecimal, octal și hexazecimal

```
#include <iostream.h>
```

```
int main()  
{  
    int numar;  
    cout << "Introdu un numar zecimal: ";  
    cin >> numar; // citirea unui număr zecimal  
    cout << "valoarea in octal = 0" << oct << numar << endl; // afișarea numărului în octal  
    cout << "valoarea in hex = 0x" << hex << numar << endl; // afișarea numărului în  
    // hexazecimal  
    cout << "Introdu un numar hexazecimal: ";  
    cin >> hex >> numar; // citirea unui număr hexazecimal  
    cout << "valoarea in octal = 0" << oct << numar << endl; // afișarea numărului în octal  
    cout << "valoarea in zecimal =" << dec << numar << endl; // afișarea numărului în zecimal  
    cout << "Introdu un numar octal: ";  
    cin >> oct >> numar; // citirea unui număr octal  
    cout << "valoarea in zecimal=" << dec << numar << endl; // afișarea numărului în zecimal  
    cout << "valoarea in hex = 0x" << hex << numar << endl; // afișarea numărului în  
    // hexazecimal  
}
```

## Exerciții:

1. Să se creeze un fișier cu extensia “.h ” în care să se declare funcții cu diferite prototipuri, cu liste de parametri și valori returnate de tip void, char, int sau float. Funcțiile vor fi definite într-un fișier cu extensia “.cpp” ce include fișierul header definit anterior, ele afișând numele funcției, valorile argumentelor și tipul returnat. Un al treilea fișier, tot cu extensie

“.cpp”, va include headerul definit și va conține funcția main() ce va apela toate funcțiile definite. Să se compileze și să se execute programul.

2. Ce vor afișa următoarele secvențe de program:

a. #include<iostream.h>

```
void main ()
{ int x=5,y;
  x=x<<3;
  cin>>y;
  cout<<(x+y);}
```

c. #include<iostream.h>

```
void main()
{ char *p;
  p="salut";
  cout<<p<<" salut";}
```

b. #include<iostream.h>

```
void main()
{ int x,y;
  cin>>x>>y;
  x=x/y;
  cout<<"x="<<x;}
```

d. #include<iostream.h>

```
void main()
{ int x=2,y=7,z=4;
  char w='f';
  cout<<x<<y<<" "<<z;
  cout<<"\n"<<w;}
```

3. Care va fi valoarea finală a variabilei "j", în urma rulării secvențelor de mai jos:

a. ....  
int i=3;  
int &j=i; j++;  
i=7; j--;  
.....

b. ....  
float i=3.14;  
int &j=i;  
j+=3;i-=1;  
j++;  
.....

4. Care va fi valoarea finală a variabilei "i", în urma rulării secvențelor de mai jos:

a. ....  
const int i=3;  
int &j=i; j++;  
.....

b. ....  
float i=3.14;  
int &j=i;  
j+=3;i-=1;  
j++;  
.....

## Fundamentele limbajului C++. Partea II.

### Operatori și expresii

Expresiile sunt componente importante în alcătuirea oricărui program. Expresiile sunt combinații de operanzi (date) și operatori. C++ posedă toți operanzii limbajului C, completând lista cu operanzi proprii: operatorii **new** și **delete** utilizați pentru alocarea dinamică de memorie, operatorul de scop ( **::** ), operatorul pointer la membru ( **.\*** ) și forma sa echivalentă ( **->.** ).

Operatorii limbajului C++ sunt prezentați în tabelul următor:

**Tabelul nr. 5.** Operatorii limbajului C++

[ ]	( )	.	->	++	--
&	*	+	-	~	!
/	%	<<	>>	<	>
<=	>=	==	!=	^	
&&		? :	=	* =	/ =
% =	+ =	- =	<< =	>> =	& =
^ =	=	,	#	##	
sizeof					
new	delete	::	.*	->*	

**Observație:** Operatorii # și ## aparțin preprocesorului.

Nivelurile de prioritate și ordinea de evaluare în cazul în care operatori consecutivi intră în alcătuirea aceleiași expresii sunt prezentate în Tabelul nr. 6.

**Tabelul nr. 6** Prioritatea și ordinea de evaluare a operatorilor

Clasa de prioritate	Operatori	Asociativitate
1	() [] - :: .	de la stânga la dreapta
2	! ~ + - ++ -- & * (tip) sizeof new delete	de la dreapta la stânga
3	. * ->*	de la stânga la dreapta
4	* / %	de la stânga la dreapta
5	+ -	de la stânga la dreapta
6	<< >>	de la stânga la dreapta
7	< <= > >=	de la stânga la dreapta
8	== !=	de la stânga la dreapta
9	&	de la stânga la dreapta
10	^	de la stânga la dreapta
11		de la stânga la dreapta
12	&&	de la stânga la dreapta
13		de la stânga la dreapta
14	?: (operator condițional)	de la dreapta la stânga
15	= *= /= %= += -= &= ^=  = <<= >>=	de la dreapta la stânga
16	, (operator virgulă)	de la stânga la dreapta

Pentru a modifica ordinea de evaluare specificată în Tabelul nr. 6 se pot utiliza parantezele ( ) care vor impune ordinea grupării operanzilor și operatorilor dorită de utilizator în evaluarea expresiilor.

În funcție de numărul de operanzi cărora li se aplică, operatorii pot fi clasificați în operatori unari (cu un operand), operatori binari (cu doi operanzi) și ternari (cu trei operanzi).

## Operatori aritmetici

- operatorul unar - : se aplică unei expresii în vederea schimbării semnului acesteia;
- operatorii binari +, -, \*, / : se aplică la doi operanzi de tip numeric;
- operatorul binar %: produce restul împărțirii a două date întregi, deci nu se poate aplica la operanzi float sau double.

**Observație:** Împărțirea întregilor produce un întreg prin trunchierea părții fracționare a rezultatului.

```
int a=2, b=3, c;  
c= 2*(a/b+1);           // variabila c primește valoarea 2  
c=a%b;                  // variabila c primește valoarea 2  
c=b%a;                  // variabila c primește valoarea 1
```

## Operatori relaționali și logici

Aceștia sunt:

- operator unar de negație: ! ;
- operatori relaționali (binari) : >, >=, <, <=;
- operatori de egalitate (binari): ==, !=;
- operatori logici (binari): &&, ||.

```
if ( !(a<=0 || b<=0)  
    return (c && d);  
else  
    return (c || d);
```

## Operatori de incrementare și de decrementare

Operatorii de incrementare (++) și de decrementare (--) sunt operatori unari și au ca efect adunarea cu 1, respectiv scăderea, operandului său. Aceștia pot fi prefixați sau post fixați. În primul caz, modificarea valorii operandului se face înainte de a-i folosi valoarea, în al doilea caz se folosește valoarea inițială a operandului și apoi se efectuează modificarea acesteia.

```
int i=5, j;  
j = ++i           // i=6, j=6;  
j = i-- ;         // i=5, j=6;
```

Acești operatori se pot aplica numai variabilelor.

```
++5 ;             // eroare, operatorul de incrementare se aplică unei constante  
(i+j)-- ;        // eroare, operatorul de decrementare se aplică unei expresii
```

## Operatori asupra biților

Acești operatori nu se pot aplica tipurilor float și double.

~	unar	complement față de 1
&	binar	ȘI bit cu bit
	binar	SAU bit cu bit
^	binar	SAU EXCLUSIV bit cu bit
<<	binar	deplasare la stânga
>>	binar	deplasare la dreapta

```
int a=7, b=2, c;  
~ a;           // expresia returnează valoarea -8  
a&b;           // expresia returnează valoarea 2  
a|b;           // expresia returnează valoarea 7  
a^b;           // expresia returnează valoarea 5  
a<<b;         // expresia returnează valoarea 28  
a>>b;         // expresia returnează valoarea 1
```

## Operatori de atribuire

Expresii de tipul:

**e1 = (e1) operator (e2);**

pot fi scrise într-o formă condensată :

**e1 operator= e2;**

unde (operator=) este operator de asignare în care operator poate fi unul dintre:

**+ - \* / % << >> & ^ |**

i = i+2;	echivalent cu	i+=2;
x= x*(y+1);	echivalent cu	x*=y+1;
a=a>>b;	echivalent cu	a>>=b;

## Operator condițional

Operatorul condițional este operatorul ternar `? : .` Se folosește în expresii sub forma:

**`e1 ? e2 : e3;`**

Expresia `e1` este evaluată prima. Dacă valoarea acesteia este logic adevărată (nonzero), se evaluează expresia `e2` și aceasta este valoarea returnată de expresia condițională, în caz contrar, se evaluează expresia `e3`, valoarea acesteia fiind cea returnată.

```
|| z=(a>b) ? a : b;           // z primește ca valoare maximul dintre a și b
```

Expresia anterioară este echivalentă cu secvența:

```
|| if (a>b)
||     z=a;
|| else
||     z=b;
```

## Alocarea dinamică de memorie folosind operatorii `new` și `delete`

C++ introduce o nouă metodă pentru alocarea dinamică a memoriei adaptată programării orientate către obiecte.

Alocarea memoriei se face cu operatorul unar **`new`**, folosind următoarea sintaxă:

```
var_ptr = new tip_var;           // 1
var_ptr = new tip_var(val_init)  // 2
var_ptr = new tip_var[dim1][dim2]...[dimN]  // 3
```

unde:

- `var_ptr` = o variabilă pointer de un tip oarecare;
- `tip_var` = tipul variabilei dinamice `var_ptr`;
- `val_init` = expresie cu a cărei valoare se inițializează variabila dinamică;

Varianta 1 alocă spațiu de memorie corespunzător unei date de tip `tip_var`, fără inițializare.

Varianta 2 alocă spațiu de memorie corespunzător unei date de tip `tip_var`, cu inițializare.

Varianta 3 alocă spațiu de memorie corespunzător unei tablou cu elemente

de tip `tip_var` de dimensiune `dim1*dim2*...*dimN`. Inițializarea tabloului nu este posibilă.

Dacă alocarea de memorie a reușit, operatorul `new` returnează adresa zonei de memorie alocate. În caz contrar, returnează valoarea `NULL` (`=0`) (în cazul în care memoria este insuficientă sau fragmentată).

```
int n=3, *p1, *p2, *p3, *p4;
p1=new int;           // variabilă întreagă neinițializată
p2=new int(15);       // variabilă întreagă inițializată cu valoarea 15
p3=new int[n];        // tablou unidimensional int de dimensiune n
p4=new int[2][n];     // tablou bidimensional int de dimensiune 2*n
```

În exemplul următor se fac alocări repetate de memorie până când se epuizează întregul spațiu din memorie:

```
#include <iostream.h>
#include <stdlib.h>

void main()
{
    double *var_ptr;
    long dim;
    cout<<"\n Introduceți dimensiunea blocului de memorie:";
    cin>>dim;
    for(int i=1; ; i++)          // în absența condiției de ieșire din ciclul for, ieșirea se va face
                                // forțat prin funcția exit() în momentul în care alocarea de
                                // memorie nu se mai poate face, deci variabila pointer var_ptr
                                // va primi valoarea 0 (NULL)
    if (var_ptr=new double[dim])
        cout<<"\n Alocare bloc nr. "<<i;
    else
    {
        cout<<"\nAlocare imposibila";
        exit(1);
    }
}
```

Eliminarea variabilei dinamice și eliberarea zonei de memorie se face cu operatorul **delete**, folosind sintaxa:

**delete var\_ptr;**

unde `var_ptr` conține adresa obținută în urma unei alocări cu operatorul `new`.



Utilizarea altei valori este ilegală, putând determina o comportare a programului nedefinită.

```
int *p;
p=new int(10);           // se alocă spațiul de memorie necesar unui int și se face
                          // inițializarea cu valoarea 10
delete p;                // se elimină variabila p din memorie
```

```
int *p,*q;
p=new int(7);            // se alocă spațiul de memorie necesar unui int și se face
                          // inițializarea cu valoarea 7
q=p;                     // variabila pointer q primește ca valoare adresa conținută în
                          // variabila p
delete q;                // se eliberează zona de memorie de la adresa conținută în
                          // variabila q
*p=17;                   // incorect, folosește o zona de memorie care a fost deja
                          // dealocată
```

```
int x=7;
int *y=&x;
delete y;                 // incorect, se cere dealocarea unei zone de memorie care nu a
                          // fost alocată cu operatorul new
```

## Operatorul de rezoluție (::)

În C++ este definit **operatorul de rezoluție (::)**, numit și operator de acces (de domeniu, scop), care permite accesul la un identificador cu domeniul fișier dintr-un bloc în care acesta nu este vizibil datorită unei alte declarații.

```
char sir[]="Sir global"; // se declară variabila globală sir
```

```
void fct()
{
    char *sir
    sir="Sir local";      // se declară variabila locală sir
    puts(::sir);          // se afișează șirul global
    puts(sir);            // se afișează șirul local
}
....
```

```
#include<iostream.h>
```

```
int a=3;                 // se declară variabila globală a cu inițializare
```

```
void f(int);

void main()
{
    int a=10;           // se declară variabila locală funcției main() cu inițializare
    f(a);               // se apelează funcția f() cu argumentul efectiv variabila locală
                        // funcției main(), a
    cout<<(::a);        // se afișează variabila globală a
}

void f(int a)           // definirea funcției cu parametrul formal a
{ a++;                 // se incrementează variabila locală funcției f()
  (::a)++;              // se incrementează variabila globală
}
```

## Conversii de tip în expresii

Conversiile de tip se pot realiza implicit sau explicit.

Dacă într-o expresie apar operanzi de tipuri numerice diferite, ei se convertesc pentru fiecare operație. În ordinea efectuării operațiilor se face în prealabil o conversie, astfel încât ambii operanzi să fie de același tip.

Regula de bază constă în conversia operandului de tip cu domeniu de valori mai mic către tipul cu domeniul de valori mai mare al celuilalt operand. Rezultatul fiecărei operații este de tipul stabilit pentru operanzi.

Conversii de tip se fac și în cazul atribuirilor. Valoarea membrului drept este convertită la tipul celui stâng. În această situație se pot efectua conversii de la un tip cu reprezentare pe un număr mai mare de octeți la un tip cu reprezentare pe număr mai mic de octeți, ceea ce poate provoca trunchieri (ex.: float ->int), rotunjiri (ex: double->float) ale valorilor sau pierderea biților de ordin superior în exces (ex: long int->char).

```
int i=2;
float r=3.5;           // valoarea constantă 3.5, care are reprezentare double, este convertită la
                        // tipul float și apoi este atribuită variabilei r
(i+2.1)*r              // 2.1 este de tip double, deci valoarea de tip int a variabilei i se
                        // convertește la tipul double, rezultatul returnat de operația i+2.1 este
                        // de tip double, ca urmare și valoarea variabilei float r se convertește la
                        // double, rezultatul final al expresiei fiind și el de tip double
```

Conversia explicită de tip se poate utiliza în orice expresie utilizându-se o construcție numită **typecast** în construcții de forma:

**(nume\_tip) expresie**

sau

**nume\_tip (expresie)**

```
int i=11;
printf("%d", i/2);           // afișează 5 – ambii operanzi sunt de tip int, deci și
                             // valoarea returnată este de tip int
printf("%f", (float)i/2);    // afișează 5.5 – se impune conversia operandului i la
                             // tipul float, deci și operandul 2 se va converti la același
                             // tip și în final tipul valorii returnate este tot float
printf("%f", i/2);           // afișare eronată – se încearcă afișarea unei valori de tip
                             // int în format float
```

## Instrucțiuni

O instrucțiune este o expresie care se încheie cu punct și virgulă (;).

Instrucțiunile se pot scrie pe mai multe linii de program, spațiile ne semnificative fiind ignorate.

```
int i;
i=
5;
```

Pe o linie de program se pot scrie mai multe instrucțiuni.

```
int i, j;
i=2; j+=i;
```

## Instrucțiunea vidă

Instrucțiunea vidă are forma:

;

și este utilizată pentru a evita utilizarea unei etichete sau atunci când corpul unui ciclu nu conține nici o instrucțiune.

## Instrucțiunea compusă (blocul de instrucțiuni)

Instrucțiunea compusă grupează declarații și instrucțiuni. Blocul de

instrucțiuni este sintactic echivalent cu o instrucțiune. Forma generală este:

**{ lista\_declaratii;  
lista\_instrucțiuni;}**

Toate variabilele declarate în interiorul unei instrucțiuni compuse nu pot fi accesate în afara acesteia, domeniul lor de existență fiind blocul de instrucțiuni.

```
#include <stdio.h>

void main()
{
    int i;                // se declară i
    i=7;
    {                    // începutul blocului
        int i;           // se redefinește i
        int j;           // se declară j
        i=9;
        j=2*i;
        printf("\ni=%d", i); // se afișează i=9
    }                    // sfârșitul blocului
    printf("\ni=%d", i);    // se afișează i=7
    printf("\nj=%d", j);    // eroare, instrucțiunea se află în afara domeniului de
                          // existență a lui j
}
```

Instrucțiunile compuse se utilizează deseori în combinație cu instrucțiunile de ciclare și cele condiționale.

### **Instrucțiunea condițională if, if-else**

Sintaxa instrucțiunii poate fi:

**if(expresie) instrucțiune\_1**

sau

**if(expresie) instrucțiune\_1 else instrucțiune\_2**

Instrucțiunea if evaluează expresia “expresie”. În cazul în care valoarea acesteia este nenulă, se execută instrucțiune\_1, iar dacă este nulă se execută instrucțiunea imediat următoare instrucțiunii if, pentru prima formă a sintaxei sau instrucțiune\_2 în cazul celei de a doua forme.

```
if (a>b)
    cout<<"a>b\n";
else
    if (a==b)
        cout <<"a=b\n";
    else
        cout<<"a<b\n";
```

### Instrucțiunea de ciclare while

Forma instrucțiunii este:

**while (expresie) instrucțiune;**

Atât timp cât "expresie" este nenulă, se execută "instrucțiune". Evaluarea expresiei are loc înainte de execuția instrucțiunii.

```
.....
i=n;
while (i)
{
    cout<<"ni="<<i;
    i--;
}
```

### Instrucțiunea de ciclare do-while

Forma instrucțiunii este:

**do instr while (expr);**

Instrucțiunea instr se execută atât timp cât expr este nenulă. Evaluarea expresiei expr se face după executarea instrucțiunii instr.

```
.....
i=n;
do
{
    cout<<"ni="<<i;
    i--;
}
while (i);
```

## Instrucțiunea de ciclare for

Sintaxa generală a instrucțiunii for este:

```
for( expr1; expr2; expr3)  
    instrucțiune;
```

- expr1 se evaluează o singură dată, la intrarea în bucla for;
- expr2 se evaluează înaintea fiecărei iterații și reprezintă condiția de ieșire din ciclu;
- expr3 se evaluează la sfârșitul fiecărei iterații, pentru actualizarea parametrilor ciclului.

```
for(i=n; i; i--)  
    cout<<"\ni="<<i;
```

## Instrucțiunea switch

Instrucțiunea switch este o instrucțiune decizională, ea permițând selecția, în funcție de valoarea unei expresii, între mai multe opțiuni posibile. Sintaxa instrucțiunii este:

```
switch (expr)  
{  
    case const1: lista_instructiuni;  
                <break;>  
    case const2: lista_instructiuni;  
                <break;>  
    .....  
    <default> lista_instructiuni;  
}
```

unde:

- expr este o expresie întreagă (orice tip întreg sau enumerare);
- const1, const2,... sunt constante de selecție, de tip întreg, cu valori distincte;
- lista\_instructiune este o secvență de instrucțiuni;

Instrucțiunea break întrerupe execuția instrucțiunii switch. Dacă lista\_instructiuni nu este urmată de break, se continuă execuția cu lista de instrucțiuni atașată următoarei etichete.

```
char cifra;  
int i;  
...  
switch (cifra)  
{  
    case '0': i=0; break  
    case '1': i=1; break  
    ...  
    case 'F': i=15; break  
}
```

### **Instrucțiunea break**

Instrucțiunea are forma:

**break;**

și are ca efect terminarea execuției unui ciclu de tip while, do-while, for sau a instrucțiunii switch, controlul fiind transferat instrucțiunii imediat următoare.

### **Instrucțiunea continue**

Instrucțiunea are forma:

**continue;**

și are ca efect trecerea la următoarea iterație într-un ciclu while, do-while sau for.

### **Instrucțiunea return**

Formele admise sunt:

**return;**

sau

**return (expr);**

sau

**return expr;**

Efectul instrucțiunii este de a trece controlul la funcția care a apelat funcția ce conține instrucțiunea return, fără transmiterea unei valori în prima formă și cu transmiterea valorii expr în celelalte.

## Instrucțiunea goto

Instrucțiunea **goto** are ca efect saltul la eticheta specificată. Sintaxa sa este:

**goto et;**

Eticheta et este specificată folosind sintaxa:

**et :**

**Observație:** Instrucțiunea goto încalcă principiile programării structurate și reduce claritatea programului, astfel încât este recomandată evitarea ei.

## Exerciții:

1. Să se definească o funcție care inversează un șir de caractere.
2. Să se scrie o funcție htoi() care convertește un șir de cifre hexazecimale în valoarea sa întreagă echivalentă. Cifrele sunt de la 0 la 9, iar literele de la a la f și de la A la F.
3. Ce afișează secvențele:

a. #include <iostream.h>

```
int a=2;
```

```
void main ()
```

```
{  
    a=3;  
    a++;  
    cout<<(::a);  
}
```

b. #include<iostream.h>

```
int a=3;
```

```
int f(int);
```

```
void main()
```

```
{ int a=5; a++;  
  cout<<"\t"<<f(a);}
```

```
int f(int a)
```

```
{ a++; cout<<(::a);  
  return a; }
```



4. Ce se întâmplă la compilarea secvențelor:

a. ....

```
int *p;  
p=new int[7];  
.....
```

c. ....

```
int *p;  
p=new int(10);  
delete p;  
.....
```

b. ....

```
int *p,*q;  
p=new int(7);  
q=p;  
delete q;  
*p=17;  
.....
```

d. ....

```
int x=7;  
int *y;  
delete y;  
*y=7;  
.....
```

# Funcții în C/C++

## Funcții

În limbajul C++ (similar cu limbajul C standard) programul este o colecție de module distincte numite funcții, structura generală a programului fiind:

```
<directive preprocesor>  
<declarații globale>  
funcții
```

Un program C++ conține obligatoriu o funcție numită **main()** și aceasta este apelată la lansarea în execuție a programului.

Programul sursă poate fi partiționat în mai multe fișiere grupate într-un proiect (se utilizează meniul Project din mediul de programare utilizat). Fiecare fișier conține declarații globale și un set de funcții, dar numai unul conține funcția main().

În C++, ca și în C, se utilizează declarații și definiții de funcții.

## Definiții de funcții

Sintaxa definiției unei funcții este:

```
tip_rez nume_funcție (<lista_parametri>)  
{  
    declarații locale  
    secvență de instrucțiuni  
}
```

unde:

- tip\_rez este un tip oarecare de dată și reprezintă tipul rezultatului returnat de funcție. Dacă nu este specificat, implicit tipul rezultatului returnat este int. Pentru funcțiile care nu returnează rezultat trebuie să se specifice tipul void.
- nume\_funcție este un identificator.
- lista\_parametri reprezintă enumerarea declarațiilor parametrilor sub forma:

**tip\_parametru nume\_parametru, <tip\_parametru nume\_parametru>**

Tipul parametrului poate fi orice tip valid de date.

Nu este admisă definirea unei funcții în blocul altei funcții și nu sunt permise salturi cu instrucțiunea goto în afara funcției.

Apelul funcției constă din numele funcției urmat de lista de constante, variabile sau expresii asociate parametrilor încadrată între paranteze ().

Atât la definirea, cât și la apelul funcțiilor, parantezele () urmează întotdeauna numele funcțiilor, chiar dacă acestea nu au parametri.

Se folosește denumirea de **parametri formali** pentru identificatorii din lista de argumente din definiția funcției și **parametri efectivi** constantele, variabilele, expresiile din lista unui apel al funcției.

Parametrii formali reprezintă variabile locale care au domeniu funcția și timpul de viață corespunde duratei de execuție a funcției, deci valorile lor se memorează în stivă sau în registrele procesorului.

## Declarații de funcții. Prototipuri

Apelul unei funcții nu se poate face înainte de definiția ei. Sunt situații în care nu este posibil acest lucru (în cazul funcțiilor care se apelează unele pe altele, sau al funcțiilor definite în alte fișiere sursă).

Pentru a oferi compilatorului posibilitatea de a verifica corectitudinea apelului unei funcții (numărul și tipurile parametrilor, tipul rezultatului, eventualele conversii care apar), se pot folosi declarații fără definire, numite **prototipuri**. Sintaxa generală a unui prototip este :

**<tip> nume\_funcție (<lista\_parametri>);**

Lipsa lista\_parametri este interpretată în C++ ca funcție fără parametri, deci cuvântul void nu este necesar, spre deosebire de C standard care nu consideră funcția fără parametri, ci cu orice listă de parametri și, ca urmare, nu verifică parametrii efectivi la apelare.

În declarația funcției este suficient ca în lista\_parametri să se specifice

tipurile parametrilor, fără identificatorii lor. Dacă însă se specifică identificatorii, aceștia trebuie să fie identici cu cei folosiți în antetul definiției funcției.

Prototipul funcției trebuie să fie plasat înainte de orice apel al funcției.

## Transferul parametrilor

### Transferul prin valoare. Conversii de tip

La apelul unei funcții, valoarea parametrilor efectivii este încărcată în zona de memorie corespunzătoare parametrilor formali. Acest procedeu se numește **transfer prin valoare**. Dacă parametrul efectiv este o variabilă, ea nu este afectată de nici o operație asupra parametrului formal, ceea ce poate constitui o protecție utilă.

Transferul de valoare este însoțit de eventuale conversii de tip realizate de compilator, implicite sau explicite dacă se folosește operatorul cast.

### Transferul prin referință

Pentru ca o funcție să poată modifica valoarea unei variabile folosită ca parametru efectiv, trebuie folosit un parametru formal de tip pointer, iar la apelul funcției să se folosească ca parametru explicit adresa variabilei.

```
#include <stdio.h>

void schimba(int *, int *);    // lista de parametri este formată din pointeri la int

void main()
{
    int i, j;
    scanf("%d%d", &i, &j);
    schimba(&i, &j);           // la apelul funcției, parametrii efectivii sunt adrese de variabile
    printf("\ni=%d, j=%d", i, j);
}

void schimba(int *a, int *b)
{
    int temp ;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

Parametri tablou constituie o excepție de la transferul parametrilor prin valoare, deoarece funcția primește ca parametru adresa tabloului. Acest lucru este motivat de faptul că, în general, tablourile conțin o cantitate mare de date al căror transfer prin valoare ar avea ca efect o scădere a vitezei de execuție și creștere a memoriei necesare prin copierea valorilor într-o variabilă locală. De altfel, numele tabloului este echivalent cu adresa sa. În prototip și antetul funcției, parametrul tablou se specifică în lista de parametri sub forma :

**tip nume\_tablou[]**

sau

**tip \*nume\_tablou**

### Transferul prin variabile referință

Folosirea parametrilor formali variabile referință este similară folosirii parametrilor formali pointeri, asigurând posibilitatea de a modifica valorile parametrilor efectivi.

Utilizarea referințelor prezintă avantajul că procesul de transfer prin referință este efectuat de compilator în mod transparent, scrierea funcției și apelul ei fiind simplificate.

Dacă tipul parametrului efectiv nu coincide cu cel al parametrului formal referință, compilatorul efectuează o conversie, ca în cazul transferului prin valoare. Pentru realizarea conversiei se creează un obiect temporar de dimensiunea tipului referință, în care se înscrie valoarea convertită a parametrului efectiv, parametrul formal referință fiind asociat obiectului temporar. Se pierde astfel avantajele folosirii referințelor ca parametri formali.

```
#include <stdio.h>

void schimba(int &, int &);           // lista de parametri este formată din referințe de
variabile int

void main()
{
    int i, j;
    float p, q;
    scanf("%d%d", &i, &j);
    schimba(i, j);                   // apelul funcției include ca parametri efectivi variabilele
                                     // i, j de tip int, deci nu este necesară conversia lor; pe
                                     // parcursul execuției funcției schimba() ele sunt
                                     // redenumite, a, respectiv b, inversarea valorilor având
                                     // efect asupra variabilelor din apel, i și j
}
```

```
printf("\ni=%d, j=%d", i, j);
scanf("%f%f", &p, &q);
schimba(p, q);           // apelul funcției include ca parametri efectivi variabile de tip
                          // float, deci va avea loc o conversie degradantă, implicită,
                          // float->int, care are ca urmare crearea de variabile temporare;
                          // referințele care se creează sunt pentru aceste variabile
                          // temporare, și nu pentru p și q; inversarea valorilor are efect
                          // asupra variabilelor temporare, astfel că, la ieșirea din funcție,
                          // se va constata faptul că valorile variabilelor p și q au rămas
                          // nemodificate
printf("\ni=%f, j=%f", p, q);
}

void schimba(int &a, int &b)
{
    int temp ;
    temp=a;
    a=b;
    b=temp;
}
```

Este utilă folosirea parametrilor formali referință și în situația în care parametrul are dimensiune mare și crearea în stivă a unei copii a valorii are ca efect reducerea vitezei de execuție și creșterea semnificativă a stivei. În această situație, dacă nu se dorește modificarea parametrului efectiv, acesta poate fi protejat prin folosirea modifierului `const` la declararea parametrului formal referință.

## **Rezultatul unei funcții. Instrucțiunea return**

Instrucțiunea `return` determină încheierea execuției unei funcții și revenirea în funcția apelantă.

Valoarea expresiei reprezintă rezultatul întors de funcție, deci trebuie să fie compatibil cu tipul indicat în prototip și definiție.

Dacă tipul funcției este `void`, instrucțiunea `return` este necesară doar dacă se dorește revenirea din funcție înainte de execuția întregii secvențe de instrucțiuni care alcătuiește funcția.

Transferul rezultatului se poate face utilizând toate cele trei metode: prin valoare, pointer sau referință.

În cazul transferului prin pointer sau referință, trebuie să se evite ca obiectul a cărui adresă se întoarce să fie un obiect automatic, deoarece acesta dispare odată cu terminarea execuției funcției. Adresa de memorie returnată va

corespunde unei zone de memorie care nu mai este alocată și pot apărea efecte neașteptate.

## Pointeri de funcții

Limbajul C++, ca și limbajul C, permite operarea cu variabile pointer care conțin adresa de început a codului executabil al unei funcții. Aceste variabile permit:

- transferul ca parametru al adresei funcției asociate;
- apelul funcției prin intermediul pointerului.

Declarația cu sintaxa:

**tip\_r(\*p\_fct)(<lista\_tip\_param>);**

declară p\_fct ca pointer cu tipul funcție cu rezultatul tip\_r și parametri lista\_tip\_param.

**Observație:** În C standard, lista tipurilor parametrilor poate lipsi și nu se fac verificări ale parametrilor, în timp ce în C++ lipsa listei parametrilor indică o funcție fără parametri.

```
#include <iostream.h>

int fct(int, float);                // prototip de funcție

void main()
{
    int (*p_f)(int, float)          // p_f, pointer de funcție
    int i=5, j;
    float r=3.5;
    p_f=fct;                        // se atribuie adresa funcției fct pointerului p_f
    j=p_f(i, r);                    // se apelează funcția fct prin pointerul p_f
    cout<<j<<endl;
}

int fct(int a, float b)
{
    cout<<a<<endl;
    cout<<b<<endl;
    return ( a + (int)b );
}
```

Programul următor compară două șiruri de caractere (prin funcția test()) folosind două criterii diferite de comparare exprimate prin funcțiile comp1() și respectiv comp2().

```
#include<stdio.h>
#include<conio.h>

void test(char*, char *, int(*)(char*, char*));      // funcția test() are un parametru de tip
                                                    // pointer de funcție

int comp1(char *, char *);
int comp2(char *, char *);

void main()
{
    char s1[80], s2[80];
    int (*p)(char*, char*);                        // se declară pointerul p de funcții cu prototipul :
                                                    // int functie (char *, char *)
    p=comp1;                                        // variabilei pointer p i se atribuie ca valoare
                                                    // adresa funcției comp1()

    puts("Introduceti primul sir");
    gets(s1);
    puts("Introduceti al doilea sir");
    gets(s2);
    test(s1, s2, p);                               // compară șirurile folosind funcția comp1()
    getch();
    p=comp2;                                        // variabilei pointer p i se atribuie ca valoare
                                                    // adresa funcției comp2()
    test(s1, s2, p);                               // compară șirurile folosind funcția comp2()
    getch();
}

void test(char*a, char*b, int(*comp)(char*, char*))
{
    if(!(comp)(a, b))
        printf("\ns1 egal cu s2");
    else
        printf("\ns1 diferit de s2");
}

int comp(char *s1, char *s2)                      // funcția compară primele caractere ale șirurilor
{ return(s1[0]-s2[0]); }

int comp1(char *s1, char *s2)                     // funcția compară lungimile șirurilor
{
    int i=0, j=0;
    while( s1[i] ) i++;
    while( s2[j] ) j++;
}
```



```
    return ( i - j );  
}
```

## Parametri cu valori implicite

C++ oferă posibilitatea declarării funcțiilor cu valori implicite ale parametrilor. La apelarea unor astfel de funcții, se poate omite specificarea parametrilor efectivi pentru acei parametri formali care au declarate valori implicite, transferându-se automat valorile respective:

```
#include <iostream.h>  
  
void f(int, int = 20);           // prototip de funcție cu un parametru normal și un  
                                // parametru implicit  
  
void main()  
{  
    cout<<"Apel normal : f( 5, 10)"<<endl;  
    f( 5, 10);  
    cout<<"Apel cu un singur parametru: f( 5 )"<<endl;  
    f( 5 );  
}  
  
void f(int a, int b)  
{  
    cout<<"a="<<a<<" "; b="<<b<<endl;  
}
```

Programul afișează:

```
Apel normal: f( 5, 10)  
a=5; b=10  
Apel cu un singur parametru: f( 5 )  
a=5; b=20
```

Pentru parametrii cărora nu li s-a asociat o valoare implicită, este obligatoriu să se specifice parametri efectivi la apelul funcției.

La declararea și definirea funcțiilor cu parametri implicați trebuie să se respecte următoarele reguli:

- Valorile implicite se specifică o singură dată, fie în prototip, fie în antetul definiției. De obicei acestea se declară în prototipul funcției, funcțiile putând fi definite extern programului.
- În lista de parametri, cei cu valori implicite trebuie să fie plasați la sfârșitul

listei. Această regulă permite identificarea univocă a valorilor implicite. La apelul funcției, lista de argumente efective va cuprinde obligatoriu valori pentru parametri neimplicți.

```
#include <iostream.h>

void f1(int = 7, float, char = 'X');           // eroare , se cere valoare implicită pentru
                                                // parametrul float
void f2(int , double = 7.5);                   // prototip corect
void f3(int, long = 111, double = 3.5);        // prototip corect

void main()
{
    float v = 1.5;
    f3( );                                     // apel incorect, lipsește parametru efectiv pentru parametrul int
                                                // care nu are valoare implicită
    f3( 1 );                                   // apel corect
    f3( 2, v);                                 // apel corect, se realizează conversia implicită, cu trunchiere,
                                                // float ->long
    f3( 3, 99);                                // apel corect
    f3( 4, 99, 7.8);                           // apel corect
}

void f2(int i, double q = 7.5)                 // eroare, se repetă declararea valorii parametrului
                                                // implicit double
{
    cout<<"\ni="<<i;
    cout<<"\tq="<<q;
}

void f3(int i, long j, double r)               // definiție corectă
{
    cout<<"\ni="<<i;
    cout<<"\tj="<<j;
    cout<<"\tr="<<r;
}
```

Programul afișează:

```
i=1      j=111      r=3.5
i=2      j=1        r=3.5
i=3      j=99       r=3.5
i=4      j=99       r=7.8
```

## Supradefinirea (supraîncărcarea) funcțiilor

Limbajul C++ oferă posibilitatea de a atribui unui simbol mai multe semnificații care pot fi distinse în context. Astfel, se poate folosi supradefinirea funcțiilor atunci când anumite activități trebuie efectuate pentru liste de parametri diferite, atât ca tipuri cât și ca număr.

Selectarea funcției se face în urma comparării tipurilor parametrilor efectiv cu tipurile parametrilor formali din diferitele declarații ale funcțiilor cu același nume. Selecția se realizează în mai multe etape de căutare a corespondenței optime, uneori implicând conversii de tip.

În cazul supradefinirii unei funcții cu un singur parametru, pașii urmați pentru selectarea funcției sunt:

1. inițial se caută coincidența tipurilor parametrului efectiv cu cel formal;
2. în caz de eșec al primei etape, se inițiază o căutare însoțită de conversii implicite nedegradante:
  - char, unsigned char, short -> int
  - unsigned short -> int sau unsigned int
  - float -> double
3. în caz de eșec al etapei a doua, se reia căutarea cu următoarele conversii standard:
  - tip numeric -> tip numeric (inclusiv conversii degradante cum ar fi float->int)
  - 0 -> tip numeric sau tip pointer oarecare
  - pointer oarecare -> void \*
  - pointer spre clasa derivată -> pointer spre clasa de bază
4. se admite utilizarea unei singure conversii de tip definită de utilizator.

Căutarea se oprește în momentul în care se identifică, în mod univoc, una dintre funcții. Dacă în una din etape se identifică mai mult de o soluție posibilă, ambiguitatea este semnalată de compilator printr-un mesaj de eroare.

```
#include <iostream.h>
```

```
void g(int);           // funcția 1  
void g(double);        // funcția 2
```

```
void main()  
{  
    int i = 5;  
    float r = 2.5;  
    char c = 'a';  
    long l = 10000;
```

```
g( i );           // apel funcția 1
g( r );           // apel funcția 2 cu conversie nedegradantă float ->double
g( c );           // apel funcția 1 cu conversie nedegradantă char ->int
g( l );           // eroare la compilare datorată ambiguității apărute în etapa a 3-a
}

void g(int a)
{
    cout<<"\nApel functia 1 cu argumentul: "<<a;
}

void g( double a )
{
    cout<<"\nApel functia 2 cu argumentul: "<<a;
}
```

Dacă funcția supradefinită are mai mulți parametri, se aplică pentru fiecare, succesiv setul de reguli descris, alegându-se în final, dacă este unică, corespondența care este superioară față de celelalte, pentru fiecare argument în parte.

```
#include <iostream.h>

void g( int, float );           // funcția 1
void g( float, int );          // funcția 2

void main()
{
    int i = 5, j = 20;
    double r = 2.5;
    char c = 'a';
    g( i, r );                  // apel funcția 1- pentru i corespondența este exactă (etapa 1),
                                // iar pentru r este necesară conversie nedegradantă (etapa 3)
    g( r, c );                  // apel funcția 2 cu conversie degradantă double -> float pentru r
                                // (etapa 3) și o conversie nedegradantă char ->int pentru i
                                // (etapa 2)
    g( i, j );                  // eroare la compilare datorată ambiguității
}

void g(int a, float b)
{
    cout<<"\nApel functia 1 cu argumentele: "
    cout<<"a="<<a<<"\tb="<<b;
}

void g(float a, int b)
{
```

```
cout<<"\nApel functia 2 cu argumentele: "  
cout<<"a="<<a<<"\tb="<<b;  
}
```

**Observație:** În procedura de selecție a variantelor funcției nu intervine tipul returnat, ci doar setul de parametri, astfel că redefinirea funcției cu tipuri returnate diferite, dar cu același set de parametri va avea ca efect eroare de compilare, ca urmare a ambiguității create.

Funcțiile cu parametri cu valori implicite sunt tratate ca funcții supradefinite cu număr crescător de parametri.

```
#include <iostream.h>  
#include<string.h>  
  
int aduna(int , int );           // 1  
double aduna(double , double ); // 2  
char *aduna(char *, char *);    // 3  
  
void main()  
{  
    int i = aduna(42, 17);       // se apelează prima funcție  
    double d = aduna(42.5, 17.9); // se apelează funcția 2  
    char * s1 = "C++ ";  
    char * s2 = "is the best!";  
    char * s = aduna(s1,s2);     // se apelează funcția 3  
    cout<<"\ni = "<<i;  
    cout<<"\nd = "<<d;  
    cout<<"\ns = "<<s;  
}  
  
int aduna(int a, int b)  
{ return a+b; }  
  
double aduna(double a, double b)  
{ return a+b; }  
  
char *aduna(char *a, char *b)  
{return strcat( a, b );}
```

## Funcții inline

Directiva de preprocesare `#define` oferă posibilitatea utilizării macrodefinițiilor cu parametri. Pentru fiecare apel, preprocesorul inserează în textul programului, în locul apelului, textul macrodefiniției în care se substituie numele parametrilor formali cu cele ale parametrilor efectivi. Compilatorul preia apoi textul, ignorând existența macrodefinițiilor.

```
#define fct( a, b ) a+b

void main()
{
    int i = 2, j = 5, k ;
    k = fct( i, j );
    float r = 2.3, s = 5.2, t;
    t = fct( r, s );
    cout<<"\nk="<<k;
    cout<<"\nt="<<t;
}
```

Avantajul folosirii macrodefiniției față de funcție este obținerea unui timp de execuție mai scurt, deoarece codul este inserat efectiv în locul apelului, eliminându-se operațiile specifice apelului de funcții ( transfer de parametri prin stivă, salvări, etc.).

Folosirea macrodefinițiilor implică însă o serie de dezavantaje:

- dimensiunea codului generat crește deoarece se inserează codul corespunzător macrodefiniției pentru fiecare apel;
- macrodefinițiile nu pot fi compilate separat;
- nu se fac verificări referitoare la tipul și numărul parametrilor;
- pot apare probleme legate de transferul parametrilor, domeniul variabilelor, etc.

```
# define fct( x ) x*x

void main()
{
    int i = 5, j;
    j=fct( i );           // corect, j = 5*5 = 25
    j=fct( i+1 );        // incorect, j= 5 + 1 * 5 + 1 = 11
}
```

Eroarea poate fi evitată definind `fct(x)` ca în exemplul următor:

```
# define fct( x ) ( x ) * ( x )

void main()
{
    int i = 5, j;
    j=fct( i );           // corect j = ( 5 ) * ( 5 ) = 25
    j=fct( i+1 );         // corect j = ( 5+1 ) * ( 5+1 ) = 36
}
```

Și în această situație pot exista apeluri care să genereze erori:

```
j = fct( i++ );           // incorect, j = ( i++ ) * ( i++ ), i fiind dublu incrementat
```

C++ oferă posibilitatea declarării funcțiilor “**inline**” care îmbină avantajele utilizării macrodefinițiilor cu cele ale utilizării funcțiilor.

Funcțiile inline păstrează toate proprietățile funcțiilor privind verificarea validității apelurilor, modul de transfer al parametrilor, domeniul declarațiilor locale.

Spre deosebire de funcțiile ordinare, la fiecare apel compilatorul inserează codul obiect al funcției inline în codul programului, ceea ce are ca efect creșterea vitezei de execuție, în schimb crește dimensiunea codului, deci funcțiile inline trebuie să fie foarte scurte. Nu se poate compila separat o funcție inline și nu se pot folosi pointeri către asemenea funcții.

Din punct de vedere sintactic, funcțiile inline se definesc și se utilizează similar celor ordinare, cu excepția adăugării specificatorului inline:

```
#include <iostream.h>

inline int max(int a, int b)
{
    return (a>b) ? a : b;
}

void main()
{
    int i, j;
    cin>>i;
    cin>>j;
    cout<<max( i, j );
}
```

Funcțiile inline nu pot conține instrucțiuni repetitive.

```
inline int* f( int, int* x)      // incorect, funcția conține instrucțiunea repetitivă for
```

```
{
  int i;
  for(i=0; i<10; i++)
    *x = 7+i;
  return x+1;
}
```

```
inline int f(int i)           // incorect, funcția conține instrucțiunea repetitivă while
{
  if( i>0 )
    while( i )
    {
      i--;
      printf("%d", i);
    }
  return 1;
}
```

```
inline int f( int x )        // definiție corectă
{ return x++ ; }
```

```
inline void f( int x )       // incorect, funcția conține instrucțiunea repetitivă do...while
{ do puts( "x" ); while( --x ); }
```

Inline este o cerere adresată compilatorului, care poate să o ignore, caz în care generează o funcție ordinară (de exemplu dacă în program se folosesc pointeri către funcția respectivă).

### Exerciții:

1. Se consideră următoarele funcții redefinite și apeluri de funcții. Care dintre acestea sunt corecte?

```
void f(int a, int b=10);
void f(int a);
...
f(20);
f(20, 30);
...
```

2. De ce nu este corectă funcția de mai jos:



```
inline int* f(int, int* x)
{ int i;
  for(i=0; i<10; i++)
    *x=7+i;
  return x+1; }
```

3. Ce afișează secvențele:

a. #include<iostream.h>

```
void f(int, int x=5);
void f(double, int y=5);

void main()
{ f(3,6); f(3.14,6); f(3); }
void f(int a, int x)
{ cout<<"\nMesaj si x="<<x; }
void f(double b, int y)
{ cout<<"\tEroare si y="<<y; }
```

b. #include<iostream.h>

```
void f(int);
void f(double);

void main()
{ f(3); f(3.14); }
void f(int x)
{ cout<<"Mesaj\t"<<x; }
void f(double y)
{ cout<<"\nEroare\t"<<y; }
```

4. Care din funcțiile de mai jos va fi apelată în funcția main()

```
int f(char){return 0;}           // Funcția a
int f(char*){return 0;}         // Funcția b
int f(int){return 0;}           // Funcția c

void main(void) { int x=f("x"); }
```

5. Care din funcțiile de mai jos pot fi declarate 'inline':

a. int f(int i)  
{ if(i>0)  
 while(i) { i--; printf("%d",i); }  
 return 1; }

b. int f(int x) { return x++;}

c. void f(int x)  
{ do puts("x"); while(--x); }

6. Redefiniți funcția de criptare a unui șir prezentată în lucrarea nr. 1, folosind parametri variabile referință.

### Clase și obiecte (partea I)

Limbajul C++ pune la dispoziția programatorului posibilitatea programării obiectuale (OOP-Object Oriented Programming). Domeniul programării orientate spre obiecte este deosebit de larg, unificând două componente de bază: datele aplicației și codul necesar tratării acestora. Se oferă facilitatea definirii unor tipuri de date proprii și a operatorilor destinați manipulării lor, acestea comportându-se asemănător datelor standard și a operatorilor standard.

Avantajele OOP se pot descrie prin următoarele concepte:

- **INCAPSULAREA** – prin care se obține contopirea datelor cu codul în cadrul așa numitelor **clase**. Se asigură o bună modularizare și localizare a eventualelor erori, precum și o bună protecție a datelor prin accesul controlat către acestea.
- **MOȘTENIREA** – care permite ca, pornind de la o clasă definită, cu anumite proprietăți, care constituie clasa de bază, să se creeze seturi de clase asemănătoare care completează proprietățile clasei de bază cu noi proprietăți.
- **POLIMORFISMUL** – într-o ierarhie de clase obținute prin moștenire, o metodă poate avea forme diferite de la o clasă la alta, utilizându-se supradefinirea acestora.

**Clasa** este un tip de date definit de utilizator care asociază unei structuri de date un set de funcții:

**Clasa = Date + Operații (metode)**

Conceptele de domeniu și durată de viață, variabile locale și globale, variabile statice, automate și dinamice se aplică și obiectelor.

C++ se distinge de limbajele POO pure prin faptul că permite controlul accesului atât la datele membre, cât și la funcțiile membre unei clase. În acest scop se folosesc specificatorii de control al accesului : **private**, **protected** și **public**.

Efectul acestor specificatori asupra accesului unui membru este:

- **public** : membrul poate fi accesat de orice funcție din domeniul de declarație a clasei;
- **private**: membrul este accesibil numai funcțiilor membre și prietene clasei;
- **protected**: membrul este accesibil atât funcțiilor membre și prietene clasei, cât și funcțiilor membre și prietene claselor derivate din clasa respectivă.

O funcție membră a unei clase are acces la toate datele membre ale oricărui obiect din clasa respectivă, indiferent de specificatorul de acces.

În C++ se pot declara mai multe categorii de clase folosind cuvintele cheie: struct, union, respectiv class. Variabilele de acest tip se numesc obiecte struct, union, respectiv class.

Tipurile class sunt mai frecvent utilizate, ele corespunzând mai fidel conceptului OOP.

## Tipul class

Sintaxa generală de declarare a unui tip de date class este similară cu a tipului struct:

```
class <nume_clasa> <: lista_clase_baza> {<lista_membri>}  
    <lista_variabile>;
```

unde:

- `nume_clasa` este un identificator care desemnează numele tipului clasă declarat, care trebuie să fie unic în domeniul în care este valabilă declarația;
- `lista_clase_baza` este lista claselor din care este derivată clasa declarată (dacă este cazul);
- `lista_membri` reprezintă secvența cu declarații de datele membre și declarații sau definiții de funcții membre; datele membre pot fi de orice tip, mai puțin tipul clasă declarat, dar se admit pointeri către acesta; folosirea specificatorilor `auto`, `extern`, `register` nu este permisă.
- `lista_variabile` este lista variabilelor de tip `nume_clasa`.

La declararea unei clase este obligatoriu să existe cel puțin una dintre `nume_clasa` și `lista_variabile`. De regulă se specifică `nume_clasa`, fapt ce permite declararea ulterioară de obiecte din tipul clasă declarat.

La declararea obiectelor de tip clasă se specifică numele clasei și lista de

variabile de acest tip.

Membrii unei clase au implicit atributul de acces private.

În cadrul declarației clasei pot să apară specificatorii de acces de oricâte ori și în orice ordine, toți membrii declarați după un specificator având accesul controlat de acesta.

Metodele asociate datelor (funcțiile membre), de regulă, trebuie să fie accesibile utilizatorului, deci se declară cu acces public, dar pot fi și situații în care să fie necesare funcții private, ele putând fi apelate doar de alte funcții membre ale clasei, sau de cele prietene clasei.

În declarația clasei se pot include doar prototipurile funcțiilor membre, definirea acestora putând fi făcută oriunde în proiect, chiar și în alt fișier.

Funcțiile membre unei clase definite în declarația acestei sunt implicit din categoria inline.

Pentru definițiile de funcții aflate în afara declarației clasei este necesar să se specifice numele clasei urmat de operatorul de rezoluție (::) alăturat numelui funcției. Operatorul indică compilatorului că funcția respectivă are același domeniu cu declarația clasei respective, fapt ce permite referirea directă a membrilor clasei. În caz contrar, compilatorul consideră că se definește o funcție cu același nume, externă clasei respective.

Funcțiile membre unei clase pot fi supradefinite și pot avea parametri implicați.

Pentru apelul funcțiilor membre publice și referirea datelor membre publice ale unui obiect, se folosesc operatorii de selecție (.) și (->), ca în cazul structurilor și uniunilor din C.

Un exemplu de clasă îl poate constitui clasa Punct care are ca date membre doi membri de tip int, x și y, ce primesc ca valori, valorile coordonatelor unui punct. De asemenea, clasa include ca funcții membre o funcție de inițializare ce stabilește valorile coordonatelor inițiale ale punctului, init(), două funcții ce permit citirea valorilor coordonatelor, getx(), respectiv gety(), funcția de modificare a coordonatelor punctului, move() și funcția de afișare, afisare(), ce afișează proprietățile punctului.

```
#include<iostream.h>

class Punct{
private:                                // se specifică accesul private la membrii clasei
    int x, y;                           // date membre ale clasei
public:                                 // se specifică acces public la membrii clasei
    void init(int initx=0, int inity=0) // funcție de inițializare, funcție membră cu
                                        // parametri implicați
        {x = initx; y = inity; }
    int getx(){ return x; }             // funcție inline, returnează valoarea membrului x
}
```

```

    int gety(){ return y; }           // funcție inline, returnează valoarea membrului y
    void move(int dx, int dy);        // funcție membră cu parametri, definită în afara
                                     // declarației clasei
    void afisare()                    // funcție de afișare a valorilor membrilor, funcție inline
    { cout<<"\nx=<<x<<"\tz="<<y";}
};

void Punct::move(int dx, int dy)      // definirea funcției move(), membră a clasei Punct
{
    x+=dx;
    y+=dy;
}

void main()
{
    Punct Punct1;                    // se declară un obiect de tip Punct, Punct1
    int x1, y1;

    cout<<"\n Introduceți coordonata x= ";
    cin>>x1;
    cout<<" Introduceți coordonata y= ";
    cin>>y1;
    Punct1.init(x1, y1);             // inițializarea obiectului Punct1 cu valorile x1,
                                     // respectiv y1
    cout<<"\n x este = "<<Punct1.getx(); // afișarea coordonatei x
    cout<<"\n y este = "<< Punct1.gety(); // afișarea coordonatei y
    Punct1.move(10, 20);             // modificarea coordonatelor obiectului Punct1
    cout<<"\n x este = "<<Punct1.getx(); // afișarea coordonatei x
    cout<<"\n y este = "<< Punct1.gety(); // afișarea coordonatei y
    Punct Punct2;                    // se declară un obiect de tip Punct, Punct2
    Punct2.init();                   // membrii x și y preiau valorile implicite ale
                                     // parametrilor, deci x=y=0
    Punct2.afisare();                // afișarea caracteristicilor obiectului Punct2
    Punct *p_Punct3;                 // se declară un obiect pointer la Punct
    p_Punct3 = &Punct2;
    p_Punct3 -> move ( 5, 12);        // apelul funcțiilor membre se face folosind
                                     // operatorul de selecție "->"
    p_Punct3 -> afisare();
}

```

Obiectele declarate de tip Punct (Punct1, Punct2) sunt structuri de date alcătuite din doi membri `int x` și respectiv `y`, care pot fi controlați cu funcțiile asociate care asigură atribuirea de valori, afișarea, modificarea acestora (`init()`, `afisare()`, `getx()`, `gety()`, `move()`).

Accesul la membrii privați `x` și `y` nu se poate face din afara clasei, ci doar prin funcțiile membre.

```
Punct1.x = 15;           // eroare, membrul Punct1.x este privat
Punct2.y = Punct1.x;     // eroare Punct1.x, Punct2.y sunt membri privați
```

Pentru fiecare obiect al clasei se alocă spațiul de memorie necesar datelor membre. Pentru funcțiile membre, în memorie există codul într-un singur exemplar, cu excepția funcțiilor inline pentru care se inserează codul pentru fiecare apel în parte.

Operatorul de atribuire ( = ) poate fi folosit pentru obiecte din același tip class, determinând copierea datelor membru cu membru, ca în cazul structurilor din C.

```
Punct2=Punct1;           //membrul x al obiectului Punct2 primește valoarea
                          // membrului x al obiectului Punct1 membrul y al
                          // obiectului Punct2 primește valoarea membrului y al
                          // obiectului Punct1
Punct2.afisare();
```

Se pot folosi operatorii new și delete pentru crearea/distrugerea de obiecte de tip class.

```
Punct * p_Punct4=new Punct;
p_Punct4 -> init(5,7);
p_Punct4-> move(2, 2);
p_Punct4-> afisare();
delete p_Punct4;
```

## Tipurile struct și union

Tipurile **struct** și **union** reprezintă în C++ cazuri particulare de clase.

Sintaxa de declarare a tipurilor struct și union este similară celei utilizate pentru class.

Tipul struct este similar tipului class, putând avea atât date membre, cât și funcții membre, dar nu asigură încapsularea implicită a datelor, implicit membrii fiind cu acces public. Pot fi utilizați specificatorii de control al accesului.

```
//clasă declarată cu struct
#include<iostream.h>

struct Punct{
private:                       // este necesar specificatorul de acces, deoarece accesul
                               // implicit este public
    int x, y;
```

```
public:
    init(int initx=0, int inity=0)
    {x=initx; y=inity;}
    int getx()
    {return x;}
    int gety()
    {return y;}
    void move(int dx, int dy);
    void afisare();
    {cout<<"\nx=<<x<<"\tz="<<y";}
};
```

În C++, tipurile union, pe lângă câmpurile de date, pot avea funcții membre, dar membrii sunt întotdeauna cu acces public, nefiind permisă utilizarea specificatorilor de acces. Tipurile union nu pot fi utilizate ca și clase de bază pentru alte clase, nici nu pot fi derivate din alte clase.

### Autoreferința. Cuvântul cheie “this”

În definițiile funcțiilor membre sunt necesare referiri la datele membre ale clasei respective. Acest lucru se poate face fără a specifica un obiect anume. La apelarea unei funcții membre, identitatea obiectului asupra căruia se acționează este cunoscută datorită transferului unui parametru implicit care reprezintă adresa obiectului.

Dacă în definiția unei funcții este necesară utilizarea adresei obiectului, acest lucru se realizează cu cuvântul cheie **this**, asociat unui pointer către obiectul pentru care s-a apelat funcția.

Clasa Punct definită anterior se poate completa cu funcția membră cu prototipul :

void adresa();

definită astfel:

```
void Punct::adresa()
{
    cout<<"\n Adresa obiectului Punct este:"
    cout<< this;
}
```

Funcția main() se poate completa cu următoarele linii de program:

```
Punct1.adresa();  
Punct2.adresa();  
p_Punct3->adresa();
```

## Constructori și destructori

Unele obiecte necesită alocarea unor variabile dinamice la creare, eventual atribuirea de valori adecvate datelor înainte de utilizare. Pe de altă parte, eliminarea unui obiect de acest tip impune eliberarea zonei de memorie alocată dinamic.

Pentru crearea, inițializarea, copierea și distrugerea obiectelor, în C++ se folosesc funcții membre speciale, numite **constructori** și **destructori**.

Constructorul se apelează automat la crearea fiecărui obiect al clasei, indiferent dacă este static, automatic sau dinamic (creat cu operatorul new), inclusiv pentru obiecte temporare.

Destructorul se apelează automat la eliminarea unui obiect, la încheierea timpului de viață sau, în cazul obiectelor dinamice, cu operatorul delete.

Aceste funcții efectuează operațiile prealabile utilizării obiectelor create, respectiv eliminării lor. Alocarea și eliberarea memoriei necesare datelor membre rămâne în sarcina compilatorului.

Constructorul este apelat după alocarea memoriei necesare datelor membre ale obiectului (în faza finală creării obiectului).

Destructorul este apelat înaintea eliberării memoriei asociate datelor membre ale obiectului (în faza inițială a eliminării obiectului).

Constructorii și destructorii se deosebesc de celelalte funcții membre prin câteva caracteristici specifice:

- numele funcțiilor constructor sau destructor este identic cu numele clasei căreia îi aparțin; numele destructorului este precedat de caracterul tilde (~);
- la declararea și definirea funcțiilor constructor sau destructor nu se specifică nici un tip de rezultat (nici tipul void);
- nu pot fi moșteniți, dar pot fi apelați de clasele derivate;
- nu se pot utiliza pointeri către funcțiile constructor sau destructor;
- constructorii pot avea parametri, inclusiv parametri implicați și se pot supradefini; destructorul nu poate avea parametri și este unic pentru o clasă.

Constructorul fără parametri se numește **constructor implicit**.

Dacă o clasă nu are constructor și destructori definiți, compilatorul



generează automat un constructor implicit, respectiv un destructor implicit, care sunt funcții publice.

Constructorii pot fi publici sau privați, dar, de regulă, se declară cu acces public. Dacă se declară constructorul private, nu se pot crea obiecte de acel tip. Acest lucru poate avea sens dacă respectiva clasă este destinată a folosi exclusiv ca și clasă de bază pentru o ierarhie de clase derivate.

## Crearea, inițializarea și eliminarea obiectelor

Se consideră inițial clasa Punct definită anterior. Pentru a asigura inițializarea automată a obiectelor de tip Punct de la declarare, se pot defini constructori care să preia operațiile efectuate de funcția init(). Clasa Punct poate fi definită astfel:

```
class Punct{
private:
    int x, y;
public:
    Punct(int initx=0, int inity=0)           // constructor cu parametri implicați
    {
        cout<<"\nConstructor Punct\n";
        x=initx; y=inity;}
    ~Punct()                                 // destructor
    { cout<<"\nDestructor Punct\n"; }
    int getx(){return x;}
    int gety(){return y;}
    void move(int dx, int dy);
    void afisare()
        {cout<<"\nx=<<x<<"\ty="<<y";}
};
void Punct::move(int dx, int dy)
{
    x+=dx;
    y+=dy;
}
```

Pentru vizualizarea apelului funcției constructor s-a inclus afișarea unui mesaj. Clasa nu are nevoie de destructor, el însă a fost definit formal pentru a permite vizualizarea momentului în care se elimină obiectele de tip Punct.

Se poate defini următorul program de testare:

```
Punct Punct1(10, 5);           // se declară obiectul Punct1, global, pentru care se preiau
                                // valorile x=10 , y=5
```

```
void main()
{
    cout << "\nIncepe functia main()";
    Punct1.afisare();
    cout << "\nSfarsit functia main()";
}
```

In urma execuției programului, se afișează:

```
Constructor Punct
Incepe functia main()
x=10 y=5
Sfarsit functia main()
Destructor Punct
```

Se observă că obiectul global Punct1 se creează înainte de a începe execuția funcției main() și se elimină după terminarea execuției acesteia. În momentul declarării obiectului Punct1 se apelează constructorul cu parametri efectivii specificați.

O declarație de forma:

```
Punct Punct2 ;
```

are ca efect apelul constructorului cu valorile implicite ale parametrilor, deci membrii obiectului Punct2 vor prelua valorile x=0, y=0.

```
class Punct{
private:
    int x, y;
public:
    Punct() // constructor implicit
    {
        cout<<"\nConstructor implicit\n";
        x=0; y=0;
    }
    Punct(int initx, int inity) // constructor cu parametri
    {
        cout<<"\nConstructor cu parametri\n";
        x=initx; y=inity;
    }
    ~Punct() // destructor
    { cout<<"\nDestructor Punct\n"; }
    int getx(){ return x; }
    int gety(){ return y; }
    void move(int dx, int dy);
}
```

```
void afisare()
{cout<<"\nx=<<x<<"\tz="<<y";}
};
```

Clasa Punct a fost definită cu supradefinirea constructorilor. În momentul creării unui obiect Punct fără specificarea parametrilor, se apelează constructorul implicit, altfel se apelează constructorul cu parametri. Destructorul este unic.

```
void main()
{
    Punct Punct1, Punct2(10, 20);
    Punct1.afisare();
    Punct2.afisare();
}
```

Programul afișează:

```
Constructor implicit
Constructor cu parametri
x=0    y=0
x=10   y=20
Destructor Punct
Destructor Punct
```

Utilizarea claselor care au definite funcții constructor și destructor garantează că obiectele create, indiferent că sunt statice, automate sau dinamice, sunt aduse într-o stare inițială adecvată utilizării, cu evitarea utilizării unor valori reziduale, iar la eliminarea lor se efectuează toate operațiile prealabile necesare.

## Constructor de copiere

La crearea unui obiect, acesta poate prelua valorile corespunzătoare ale unui obiect deja existent, prin apelul unui constructor special, numit constructor de copiere.

Declarația constructorului de copiere pentru o clasă este un constructor cu un parametru unic de tip referință la obiecte de tipul clasei ce se definește:

**nume\_clasa( nume\_clasa &);**

În absența definirii explicite a constructorului de copiere în cadrul clasei,

compilatorul generează automat un constructor de copiere care inițializează datele membre ale obiectului nou creat cu valorile corespunzătoare ale obiectului specificat.

```
Punct P1(10, 15);  
Punct P2(P1);           // se generează un constructor de copiere implicit care va face  
                        // atribuirile P2.x=P1.x, P2.y=P1.y  
P2.afisare();
```

În declarația clasei Punct se poate adăuga constructorul de copiere care va avea prototipul:

```
Punct (Punct &);
```

și se definește:

```
Punct::Punct(Punct &P)  
{  
    cout<< "\n Constructor de copiere";  
    x=P.x;  
    y=P.y;  
}
```

Se pot crea obiecte copie a unor obiecte existente folosind declarații similare exemplului anterior:

```
Punct P2(P1);
```

În cazul clasei Punct definită, nu este necesară definirea constructorului de copiere, constructorul de copiere generat de compilator putând să asigure operațiile necesare creării noului obiect. Există însă situații în care definirea acestuia este absolut necesară. Această situație apare, de regulă, atunci când clasa definită conține date membre pentru care s-a făcut alocare dinamică de memorie. Prin copierea datelor membru cu membru se poate ajunge în situația de a referii aceeași zonă de memorie prin membrii a două obiecte diferite. La eliminarea unuia dintre ele se face și dealocarea zonei respective de memorie, ea putând fi referită în continuare de obiectul încă existent. La eliminarea și a acestui obiect se va încerca eliberarea zonei de memorie deja dealocată.

Pentru ilustrarea celor spuse anterior se definește clasa “tablou” care are ca date membre un tablou de elemente float, pentru toate obiectele tablou numărul elementelor fiind același, N :

```
#include <iostream.h>
#define N 10

class tablou
{
    float tab[N];
public:
    tablou();
    void citire();
    void afisare();
    void modif(int, float);
};

tablou::tablou()                // constructor implicit - inițializează elementele tabloului
{                                // cu valoarea 0
    for (int i=0; i<N; i++)
        tab[i]=0;
}

void tablou::citire()            // funcție membră prin care se citesc de la tastatură
{                                // valorile elementelor din tab
    for(int i=0; i<N; i++)
    { cout<<"\ntab["<<i<<"]=";
      cin>>tab[i];
    }
}

void tablou::afisare()           // funcție de afișare a adresei obiectului și a valorilor
{                                // elementelor din tab
    cout<<"\nadresa="<<this;
    for(int i=0; i<N; i++)
        cout<<"\ntab["<<i<<"]="<<tab[i];
}

void tablou::modif(int i, float el) // funcție membră prin care se modifică valoarea unui
{                                    // element al lui tab
    tab[i]=el; }

void main()
{
    tablou t1;                    // se declară obiectul de tip tablou t1
    t1.citire();
    tablou t2(t1);                // se creează obiectul de tip tablou t2 prin copierea lui t1; se
    // apelează constructorul de copiere generat de compilator
    t2.modif(0, 2.5);
}
```

```
t1.afisare();
t2.afisare();
}
```

Se reia declararea clasei tablou pentru a obține tablouri de dimensiuni diferite. Este necesară alocarea dinamică a membrului tab.

```
#include <iostream.h>

class tablou
{
    int nr_el;           // dimensiunea tabloului tab
    float *tab;          // adresa tabloului tab
public:
    tablou();             // constructor implicit
    tablou(int);          // constructor cu parametru
    tablou(tablou &);     // constructor de copiere
    ~tablou();            // destructor
    void citire();
    void afisare();
    void modif(unsigned int, float);
};

tablou::tablou()          // definire constructor implicit
{
    cout<<"\nConstructor implicit";
    nr_el=10;
    tab=new float[nr_el];
    for(int i=0; i<nr_el; i++)
        tab[i]=0;
}

tablou::tablou(int n)     // definire constructor cu parametru
{
    cout<<"\nConstructor cu parametru ";
    nr_el=n;
    tab=new float[nr_el];
    for(int i=0; i<nr_el; i++)
        tab[i]=0;
}

tablou::tablou(tablou &t) // definire constructor de copiere
{
    cout<<"\nConstructor de copiere";
    nr_el=t.nr_el;
    tab=new float[nr_el];
    for(int i=0; i<nr_el; i++)
        tab[i]=t.tab[i]; }
}
```

```
tablou::~tablou()                                // definire destructor
{
    cout<<"\nDestructor" ;
    delete tab;
}

void tablou::citire()
{
    for(int i=0; i<nr_el; i++)                    // definirea funcției de citire a valorilor
                                                    // elementelor tabloului
    { cout<<"ntab["<<i<<"]=";
      cin>>tab[i];
    }
}

void tablou::afisare()                            // definirea funcției de afișare a datelor
{
    cout<<"\nadresa="<<this;
    for(int i=0; i<nr_el; i++)
        cout<<"ntab["<<i<<"]="<<tab[i];
}

void tablou::modif(unsigned int i, float el)      // definirea funcției de modificare a valorii
                                                    // elementului de index i al tabloului
{
    if (i>=nr_el)
        cout<<"\nParametru ilegal";
    else
        tab[i]=el;
}

void main()
{
    tablou t1(5);                                // se apelează constructorul cu parametru
    t1.citire();
    tablou t2(t1);                                // se apelează constructorul de copiere
    t2.modif(0, 2.5);
    t1.afisare();
    t2.afisare();
    tablou t3;                                    // se apelează constructorul implicit
    t3.citire();
    t3.afisare();
}
```

Prin alocare dinamică de memorie pentru membrul tab, tablourile pot fi create de dimensiuni diferite. Acest fapt atrage necesitatea definirii destructorului care să elibereze zona respectivă de memorie.

De asemenea, în exemplul prezentat este necesară definirea constructorului de copiere. În absența acestei definiții, constructorul de copiere

creat implicit de compilator va atribui aceeași adresă membrului tab al obiectului t2, cu cea a conținutului obiectului t1, deci ele vor referi aceeași zonă de memorie. Modificarea valorii unui element pentru t1 va fi preluată și de t2 și reciproc. În momentul distrugerii obiectelor t1 și t2 se va încerca dealocarea unei aceleiași zone de memorie.

### Exerciții :

1. Să se definească tipul de date:

```
class ceas
{ private :
    long int secunde;
public :
    ceas();
    ceas(int o, int m, int s=0);
    ceas( ceas &);
    ~ceas();
    afisare(); };
```

care reprezintă timpul scurs de la ora 0:0:0 a unei zile, exprimat în secunde. Funcțiile membre vor conține afișarea de mesaje și afișarea autoreferinței obiectelor, astfel încât să se poată urmări, la execuția programului, obiectul asupra căruia se operează și funcția apelată. Funcția main() va conține declarații de obiecte de tip ceas cu apel al diferiților constructori și atribuire între obiecte de tip ceas și afișarea proprietăților obiectelor ceas prin apelul funcției afisare().

2. Să se definească tipul de date:

```
class sir
{ private:
    char * continut;
    int dim;
public:
    sir(int d = 80);
    sir (sir &);
    ~sir();
    void citire();
    void afisare(); };
```

Funcția main() va conține declarații de obiecte sir pentru care se citește conținutul. Pentru fiecare obiect se va crea o copie ce va fi afișată.



### Clase și obiecte (partea II)

#### Manevrarea dinamică a obiectelor

În mod similar utilizării funcțiilor C malloc() și respectiv free(), se pot crea și distruge obiecte dinamice utilizând operatorii new și delete.

Operatorul new alocă memorie corespunzător tipului datei, fără a fi necesar să se specifice dimensiunea zonei de memorie ca în cazul funcției malloc(). În plus, în cazul tipurilor clasă produce apelul unui constructor, ceea ce nu se întâmplă în cazul funcției malloc().

Operatorul delete este analog funcției free(), dar, în plus față de aceasta, în momentul eliminării obiectului din memorie va apela, dacă este cazul, funcția destructor.

De exemplu, pentru clasa tablou cu declarația:

```
class tablou
{
    int nr_el;                // dimensiunea tabloului tab
    float *tab;              // adresa tabloului tab
public:
    tablou();                // constructor implicit
    tablou(int);             // constructor cu parametru
    ~tablou();               // destructor
    void citire();
    void afisare();
};
```

se pot crea obiecte dinamice astfel:

```
tablou * p_t1 = new tablou;    // se creează obiectul dinamic p_t1, cu apelarea
                                // constructorului implicit
p_t1 -> afisare();
p_t1 -> citire();
p_t1 -> afisare();
```

```
delete p_t1; // de elimină obiectul p_t1, odată cu apelul
              // destructorului
tablou * p_t2 = new tablou (3) // se creează obiectul dinamic p_t2, cu apelarea
                              // constructorului cu parametru
p_t2 -> afisare();
p_t2 -> citire();
p_t2 -> afisare();
delete p_t2; // se elimină obiectul p_t2, odată cu apelul destructorului
```

## Transferul obiectelor ca parametri sau rezultat

În expresii, operanzii pot fi obiecte ale claselor definite, la fel rezultatul returnat de către acestea. De asemenea, parametrii funcțiilor și rezultatul returnat de către acestea pot fi obiecte ale unor clase.

În procesul evaluării expresiilor, ca și la preluarea valorilor pentru parametri și returnarea rezultatelor, apar situații de memorare temporară de valori, creându-se obiecte temporare. Obiectele temporare au durata de viață limitată la durata de execuție a blocului de instrucțiuni căruia îi aparțin, dar, spre deosebire de variabilele automate, ele au existența ascunsă.

Un caz particular îl constituie parametrii transferați prin valoare. Ei pot fi asimilați cu variabilele automate, dar spre deosebire de acestea timpul lor de viață este mai lung decât timpul de execuție al domeniului lor. Aceste obiectele temporare sunt create înainte de începerea execuției funcției și sunt eliminate după încheierea execuției acesteia.

La transferul prin valoare a unui parametru sau al unui rezultat se creează obiecte temporare prin apelul constructorului de copiere al clasei respective, sau a constructorului de copiere generat de compilator, în lipsa definiției acestuia.

În exemplul următor se declară o clasă “punct” în care în funcțiile constructor și destructor se includ mesaje corespunzătoare și afișarea adresei obiectului pentru a putea urmări crearea și eliminarea obiectelor.

```
#include <iostream.h>

class punct
{
    int x, y;
public:
    punct(int=0, int=0 );
    punct(punct&);
    ~punct();
    void deplasare(int, int);
```

```
void afisare();
};

punct::punct(int abs, int ord)
{
    x=abs;
    y=ord;
    cout<<"\nConstructor "<<this;
    afisare();
}

punct::punct(punct &p)
{
    x=p.x;
    y=p.y;
    cout<<"\nConstructor copiere "<<this;
    afisare();
}

punct::~punct()
{
    cout<<"\nDestructor "<<this;
    afisare();
}

void punct::deplasare(int dx, int dy)
{
    x+=dx;
    y+=dy;
}

void punct::afisare();
{
    cout<<"\nx="<<x<<"\ty="<<y;
}

punct test(punct);

void main()
{
    cout<<"\nApel main()";
    punct p0, p1(1,1);           // p0 se creează prin apelul constructorului cu parametri
                                // cu valorile implicite, p1 preia valorile x=1, y=1
    p0=test(p1);                 // rezultatul funcției test(), p, este atribuit membru cu
                                // membru lui p0
    p0.afisare();
    cout<<"\nFinal main()";
}
```

```
punct test(punct p)           // parametrul formal p se creează apelând constructorul
                                // de copiere
{
    cout<<"\nApel test()";
    p.deplasare(10,10);
    return p;                  // rezultatul se returnează printr-un obiect temporar creat
                                // prin apelul constructorului de copiere
}
```

Programul afișează:

```
Apel main()
Constructor 0xfff2             // se creează p0
x=0    y=0
Constructor 0xffee            // se creează p1
x=1    y=1
Constructor copiere 0xffe6     // se creează parametrul ca obiect temporar – echivalent
                                // cu apelul p(p1)
x=1    y=1
Apel test()
Constructor copiere 0xffea     // se creează obiectul temporar care transferă rezultatul
x=11   y=11
Destructor 0xffe6              // se elimină p
x=11   y=11
Destructor 0xffea              // se elimină obiectul care transferă rezultatul
x=11   y=11
x=11   y=11
Final main()
Destructor 0xffee              // se elimină p1
x=1    y=1
Destructor 0xfff2              // se elimină p0
x=11   y=11
```

Pentru obiecte de dimensiuni mari, se recomandă transferul parametrilor și rezultatului prin referință deoarece nu se mai creează obiecte temporare pentru parametri, astfel încât se reduce încărcarea stivei și crește viteza de execuție.

Funcția test() se poate defini astfel:

```
punct & test(punct &p)        // parametrul formal p de definește ca referință u unui punct
{
    cout<<"\nApel test()";
    p.deplasare(10, 10);
    return p;                  // rezultatul se returnează printr-un obiect temporar creat prin
                                // apelul constructorului de copiere
}
```

Programul afișează:

```
Apel main()
Constructor 0xffff2          // se creează p0
x=0    y=0
Constructor 0xffee          // se creează p1
x=1    y=1
Apel test()
x=11   y=11
Final main()
Destructor 0xffee           // se elimină p1
x=11   y=11
Destructor 0xffff2          // se elimină p0
x=11   y=11
```

Se observă că nu se mai creează obiecte temporare pentru parametrul și rezultatul funcției, în schimb modificările care au loc asupra obiectului p din funcția test() sunt de fapt modificări ale obiectului p1 a cărui referință este.

Obiecte temporare nu se creează numai prin constructorul de copiere, ci pot apare situații în care se creează prin apelul unui constructor uzual:

```
punct p1(1, 1);
p1=punct(10, 10);
```

În această situație, se creează un obiect temporar prin apelul constructorului cu parametri punct(10, 10), după care se face copierea membru cu membru a valorilor membrilor obiectului temporar în obiectul p1.

## Clase cu membri obiecte

Clasele pot avea membri de orice tip, mai puțin tipul clasă care se definește. Deci membrii pot fi obiecte de diferite tipuri.

Se definește clasa “pozitie” care reprezintă poziția unui punct pe ecran și clasa “text\_poz” care va conține un șir de caractere asociat unei poziții specificată printr-un obiect pozitie:

```
#include <iostream.h>

class pozitie                // declarația clasei pozitie
{
    int x, y;
public:
    void init(int abs, int ord)
```

```
{ x=abs; y=ord; }
void deplasare(int abs, int ord)
{ x+=abs; y+=ord; }
void afisare()
{ cout<<"\nx="<<x<<"\ty="<<y;}
};

class text_poz {                                //declarația clasei text_poz
char *txt;
pozitie orig;                                // se declară membrul de tip pozitie ce va conține
                                              // coordonatele de afișare a textului
public:
text_poz(int , int , char*);                // constructor cu parametri
void depl_orig(int , int );
void afisare()
{ orig.afisare();
  cout<<txt;
}
};

text_poz::text_poz(int abs, int ord, char*c=NULL)
{
txt=c;
cout<<"Constructor text_poz: "<<txt<<"\n";
orig.init(abs, ord);
};

void text_poz::depl_orig(int dx, int dy)
{
orig.deplasare(dx, dy);
};

void main()
{
text_poz txt(10, 10, "text");
txt.depl_orig(5, 5);
txt.afisare();
txt.orig.deplasare(5, 5);                    // eroare, orig este membru private
}
```

La crearea unui obiect `text_poz`, pentru crearea membrului `orig`, în absența unui constructor declarat, se apelează constructorul generat de compilator clasei punct. Dacă există un constructor declarat în clasa `pozitie`, la crearea membrului `orig` se va apela constructorul respectiv. Dacă acesta este unic și este declarat cu parametri ordinari, trebuie transmise valorile corespunzătoare acestor parametri, deci un constructor generat implicit de compilator sau un constructor implicit

declarat pentru clasa care conține membru obiect nu pot rezolva această situație. Deci este necesar, fie să se definească un constructor implicit pentru clasa membrului, fie să se definească un constructor prin care să se definească transferul parametrilor către constructorul clasei membrului înglobat.

```
#include <iostream.h>

class pozitie
{
    int x, y;
public:
    pozitie();                // constructor implicit
    pozitie(int abs, int ord); // constructor cu parametri
    void deplasare(int abs, int ord);
    void afisare();
};

class text_poz {
    char *txt;
    pozitie orig;
public:
    text_poz();                // constructor implicit – la crearea unui obiect text_poz
                                // se va apela întâi constructorul implicit pentru membrul
                                // pozitie și apoi cel al clasei text_poz

    void citire_sir();
    void afisare();
    ...
};
```

Transferul parametrilor prin constructorul clasei text\_poz către constructorul clasei pozitie se realizează prin specificarea parametrilor destinați obiectului pozitie în antetul constructorului :

```
#include <iostream.h>

class pozitie
{
    int x, y;
public:
    pozitie(int abs, int ord)                //constructor cu parametri
    { x=abs; y=ord;
      cout<<"\nConstructor pozitie ";
      afisare();
    }
    ~pozitie()                               // destructor
    { cout<<"\nDestructor pozitie"; }
};
```

```
void deplasare(int abs, int ord)
{
    x+=abs;
    y+=ord;
}
void afisare()
{
    cout<<"\nx="<<x;
    cout<<",y="<<y<<"\n";
}
};

class text_poz                                //clasa cu membri obiecte
{
    char *txt;
    pozitie orig;                             //poziția textului
public:
    text_poz(int ,int , char*);               // constructor cu parametri
    ~text_poz()                               //destructor
    { cout<<"\nDestructor text_poz";}
    void depl_orig(int , int );
    void afisare()
    { orig.afisare();
      cout<<txt;
    }
};

    // se specifică parametrii preluați de constructorul pozitie la crearea membrului orig
text_poz::text_poz(int abs, int ord, char*c=NULL) : orig(abs, ord)
{
    txt=c;
    cout<<"\nConstructor text_poz: ";
    orig.afisare();
    cout<<txt<<"\n";
};

void text_poz::depl_orig(int dx, int dy)
{
    orig.deplasare(dx, dy);
};

void main()
{
    int i;
    cout<<"\nIncepe main()";
    text_poz txt(10, 10, "Text exemplu");
    txt.depl_orig(5, 5);
    txt.afisare();
}
```



```
| cout<<"\nFinal main()";  
| }
```

La execuția programului se va afișa:

```
| Incepe main()  
| Constructor pozitie:  x=10  y=10  
| Constructor text_poz: x=10  y=10  
| Text exemplu  
| x=15, y=15  
| Text exemplu  
| Final main()  
| Destructor text_poz  
| Destructor pozitie
```

Se observă că se apelează întâi constructorul `pozitie()` care creează membrul “orig” și apoi constructorul `text_poz()`. Ordinea eliminării obiectelor este inversă, întâi se distruge obiectul `text_poz` și apoi cel `pozitie`.

## Tablouri de obiecte

Tablourile pot avea elemente de orice tip, inclusiv de tip clasă.

La crearea unui tablou cu elemente de tip clasă, se va apela constructorul clasei tip element pentru fiecare element în parte. Problemele semnalate la crearea obiectelor cu membri de tip clasă vor apărea și în acest caz. Mai mult, în cazul tablourilor nu există posibilitatea specificării valorilor corespunzătoare parametrilor, deci pentru tipul clasă corespunzător elementelor tabloului este obligatoriu să existe declarat constructor implicit sau constructor cu toți parametrii implicați. Pentru fiecare element de tablou de tip clasă se apelează constructorul, la încetarea domeniului de existență a tabloului, pentru fiecare element de tablou se apelează destructorul clasei.

```
| #include <iostream.h>  
  
| class pozitie  
| {  
|     int x,y;  
| public:  
|     pozitie(int abs, int ord)                //constructor cu parametri  
|     { x=abs; y=ord;  
|         cout<<"Constructor  cu parametri - pozitie ";  
|         afisare();  
|     }  
| }
```

```
pozitie()                                //constructor implicit
{
    x=0; y=0;
    cout<<"Constructor implicit - pozitie ";
    afisare();
}

~pozitie()
{ cout<<"\nDestructor pozitie"; }

void deplasare(int abs, int ord)
{
    x+=abs;
    y+=ord;
}

void afisare()
{ cout<<"x="<<x;
  cout<<"y="<<y<<"\n";
}
};

main()
{
    int i;
    cout<<"Incepe main()\n";
    pozitie tab[10];                      //apelează constructorul fără parametri de 10 ori
    for (i=0; i<10; i++)
        tab[i].deplasare(i, i);
    for (i=0; i<10; i++)
        tab[i].afisare();
    cout<<"Final main()\n";               // la ieșirea din main() încetează domeniul de existență al
                                           // lui tab și se apelează de 10 ori destructorul clasei
                                           // poziție
}
}
```

În mod similar se întâmplă în cazul creării de tablouri prin alocare dinamică de memorie:

```
pozitie *p_poz=new pozitie[5];           // tablou alocat dinamic – se apelează constructorul fără
                                           // parametri de 5 ori

for (i=0; i<5; i++)
{
    p_poz[i].deplasare(i+2, i+2);
    p_poz[i].afisare();
};
```

```
delete [] p_poz;           // eliberarea zonei de memorie prin apelul de 5 ori a  
                           // funcției destructor
```

În exemplul anterior, o instrucțiune de forma:

```
delete p_poz;
```

are ca efect eliberarea spațiului corespunzător elementului `p_poz[0]`, spațiul corespunzător celorlalte elemente fiind în continuare ocupat. Deci sintaxa folosită pentru dealocarea completă a unui tablou (`nume_pointer`) alocat dinamic cu operatorul `new` este:

```
delete [ ] nume_pointer ;
```

## Pointeri către membrii unei clase. Operatorii ( `.*` ) și ( `->*` )

Limbajul C++ oferă posibilitatea de a opera cu pointeri către date și funcții membre ale unei clase. Spre deosebire de pointerii ordinari, aceștia au asociat, pe lângă tipul datei sau funcției, și tipul clasă respectiv. Un pointer către un membru al unei clase nu se asociază obiectelor clasei respective, ci clasei, el conținând ca informație nu adresa membrului ci deplasarea lui în cadrul clasei. Sintaxa declarației unui pointer la membrul clasei `nume_clasă` este:

```
tip nume_clasa::*pointer_membru;
```

Pentru atribuirea valorii pointerului se folosește același operator, `&`:

```
pointer_membru=&nume_clasa::membru;
```

Operatorii destinați accesului la un membru prin pointeri sunt:

- operatorul `.*` dacă se specifică un obiect al clasei
- operatorul `->*` dacă se specifică un pointer de obiect

Sintaxa folosită pentru referirea membrului este:

```
obiect.*pointer_membru
```

sau

```
pointer_obiect->*pointer_membru
```

```
#include <iostream.h>

class pozitie
{
public:
    int x, y;
    pozitie(int abs, int ord)
    { x=abs; y=ord;
      cout<<"Constructor cu parametri-pozitie ";
      afisare();
    }
    ~pozitie()
    { cout<<"\nDestructor pozitie";}
    void deplasare(int abs, int ord)
    { x+=abs; y+=ord; }
    void afisare()
    { cout<<"\nx="<<x<<"\ty="<<y; }
};

void main()
{
    pozitie p(1, 1), *p_poz;           // se declară un obiect pozitie și un pointer la pozitie
    int pozitie::p_x, pozitie::p_y;    // se declară pointeri la membri de tip întreg ai
                                        // clasei pozitie
    void (pozitie::*p_func)(int, int); // se declară pointer la o funcție membră clasei
                                        // pozitie cu prototip "void functie(int, int)"
    p_poz=&p;                           // se atribuie pentru pointerul p_poz adresa obiectului p
    p_x=&pozitie::x;                     // se atribuie pointerului p_x adresa membrului x al
                                        // obiectului p
    p_y=&pozitie::y;                     // se atribuie pointerului p_y adresa membrului y al
                                        // obiectului p
    p_func=&pozitie::deplasare;          // se asociază pointerului la funcție p_func funcției
                                        // membră clasei pozitie, deplasare()
    (p.*p_func)(5, 5);                  // se apelează funcția membră deplasare() pentru
                                        // obiectul p prin pointerul la funcție p_func
    cout<<"\nx="<<p.*p_x;                // se afișează membrul x al obiectului p prin pointerul
                                        // p_x
    cout<<"\ny="<<p.*p_y;                // se afișează membrul y al obiectului p prin pointerul
                                        // p_y
    (p_poz->*p_func)(2, 2);              // se apelează funcția membră deplasare() pentru
                                        // obiectul pointer la pozitie p_poz prin pointerul la
                                        // funcție p_func
    cout<<"\nx="<<p_poz->*p_x;            // se afișează membrul x al obiectului p_poz prin
                                        // pointerul p_x
    cout<<"\ny="<<p_poz->*p_y;            // se afișează membrul y al obiectului p_poz prin
                                        // pointerul p_y
}
```

În cazul pointerilor la funcții membre nu este obligatorie utilizarea operatorului &, deci declarațiile:

```
p_func=&pozitie::deplasare;  
și  
p_func=pozitie::deplasare;  
sunt echivalente.
```

## Membri statici ai unei clase

Membrii unei clase, date sau funcții, pot fi declarați statici.

Datele membre nestatice ale unei clase sunt distincte pentru fiecare obiect al clasei în parte, în timp ce datele statice sunt unice pentru toate obiectele clasei respective, ele fiind create, inițializate, accesate independent de obiectele clasei.

Funcțiile membre statice efectuează operații asupra întregii clase, ne fiind asociate unor obiecte individuale.

Referirea membrilor statici se poate face astfel:

- numele clasei împreună cu operatorul de rezoluție și numele membrului static, chiar dacă nu există obiecte ale clasei respective declarate:

```
nume_clasa :: nume_membru
```

- numele unui obiect al clasei împreună cu operatorul de selecție și numele membrului, asemenea membrilor nestatici:

```
nume_obiect.nume_membru
```

Alocarea memoriei și inițializarea membrilor statici se fac separat de celelalte date, fiind obligatorie existența unei definiții, eventual cu inițializare, externă clasei. Inițializarea implicită se face cu valoarea 0.

```
#include <iostream.h>
```

```
class pozitie
```

```
{
```

```
    int x, y;
```

```
    static int st;
```

```
// declarare membru static, care se va folosi ca un contor pentru
```

```
// obiectele din clasa pozitie create; definirea se face în
```

```
// exteriorul clasei
```

```
public:
```

```

pozitie(int abs, int ord)
{ x=abs; y=ord;
  st++;                                     // la crearea unui obiect pozitie, contorul st se
                                           // incrementează
  cout<<"\nS-a creat obiectul pozitie nr: "<<st<<"\t";
  afisare();
}
~pozitie()
{ cout<<"\nDestructor pozitie";
  cout<<"\nSe distruge obiectul nr. "<<st;
  st--;                                     // odată cu distrugerea unui obiect pozitie, contorul st se
                                           // decrementează
}
static void nr_obj()                       // funcție statică ce permite accesul la membrul static
                                           // privat st
{cout<<"\nNr. obiecte: "<<st;}

void deplasare(int abs, int ord)
{ x+=abs;
  y+=ord; }

void afisare()
{ cout<<"\nx="<<x<<"\ty="<<y; }
};

int pozitie::st;                          // definirea membrului static st; implicit este inițializat
                                           // cu valoarea 0 definirea cu inițializare este :
                                           // int pozitie::st=0

void main()
{
  pozitie::nr_obj();                      // apel funcție statică prin numele clasei
  pozitie::st;                            // eroare, membrul st este private
  pozitie p1(1,1), p2(2,2), *p3;
  p1.nr_obj();                            // apel funcție statică prin numele unui obiect poziție
  p3=new pozitie(3,3);
  pozitie::nr_obj();                      // apel funcție statică prin numele clasei
  delete p3;
}

```

Funcțiile statice nu primesc implicit adresa unui obiect, apelul pentru toate obiectele clasei fiind identic. Din această cauză, în funcțiile statice nu se poate folosi cuvântul `this`, iar membrii nestatici pot fi referiți doar prin numele obiectelor. Din funcțiile statice se pot referi direct doar membrii statici.

```

static int return_st()
{
  cout<<"adresa obiectului"<<this;      // eroare, nu se poate folosi autoreferința this
}

```

```
cout<<"valoare x:"<< x;           // eroare, nu se pot referi membri nestatici in mod direct
return st;
}
```

## Funcții și clase prietene unei clase

Accesul la membrii private ai unei clase se poate acorda, în afara funcțiilor membre și unor funcții nemembre, dacă sunt declarate cu specificatorul **friend**.

Funcțiile declarate prietene unei clase pot fi funcții independente sau funcții membre ale altei clase. Funcțiile prietene sunt externe clasei, deci apelul lor nu se face asociat unui obiect al clasei.

```
class cls
{
    ...
    friend void func();
}

void main()
{
    cls obj;
    obj.func();           // eroare, funcția func() este externă clasei
    func();               // apel corect al funcției func()
    ...
}
```

Funcțiile prietene au acces la toți membrii clasei, ele operând asupra obiectelor care se transferă ca parametrii ai funcției. Referirea unui membru se face prin numele obiectului, operatorul de selecție și numele membrului, dată sau funcție.

Se pot întâlni următoarele situații:

1. o funcție independentă este prietenă unei clase ;
2. o funcție membră a unei clase este prietenă altei clase;
3. o funcție este prietenă mai multor clase ;
4. o clasă este prietenă altei clase (toate funcțiile membre ale unei clase sunt prietene celeilalte clase).

// Situația 1.

```
class pozitie
{ int x, y;
```

```
public:
    pozitie(int abs, int ord)
    { x=abs; y=ord; }
    void deplasare(int dx, int dy)
    { x+=dx; y+=dy; }
    friend void compar(pozitie &, pozitie &);    // funcția compar() se declară prietenă
                                                // clasei pozitie
};

void compar(pozitie &p1, pozitie &p2)           // funcția compar() este externă clasei
                                                // poziție, deci nu poate fi definită decât
                                                // în afara clasei pozitie
{
    if ((p1.x==p2.x)&&(p1.y==p2.y))             // funcția prietenă compar() are acces la
                                                // membrii private ai clasei pozitie
        cout<<"\n Pozitii identice";
    else
        cout<<"\n Pozitii diferite";
}

void main()
{
    pozitie p1(1, 1); p2(3, 3);
    compar(p1, p2);                             // apel al funcției compar()
    p1.deplasare(2, 2);
    compar(p1, p2);                             // apel al funcției compar()
}
```

// Situația 2

```
#include <iostream.h>
class A;                                         // declarație incompletă a clasei A
class B                                         // declarația clasei B
{
    int b;
public:
    B(int n)
    {b=n;}
    void func(A &);                             // declararea funcției membre a lui B, având
                                                // parametru de tip A
};

class A
{
    int a;
public:
    A(int n)
    {a=n;}
}
```



```
friend void B::func(A &);           // se declară funcția membră clasei B func()
};                                  // prietenă clasei A

void B::func( A &ob_a)              // definirea funcției func() membră a clasei B
{
    cout<<"\n"<<ob_a.a;           // funcția fiind prietenă clasei A are acces la
                                    // membrul private "a" al clasei A
    cout<<"\n"<<b;
}

void main()
{
    A ob1(10);
    B ob2(20);
    ob2.func(ob1);                  // apelul funcției membre clasei B
}
```

Declarația friend din clasa A trebuie să găsească clasa B căreia îi aparține funcția func() declarată. În declarație trebuie să se specifice și numele clasei căreia îi aparține funcția prietenă.

```
// Situația 3

#include <iostream.h>
class A;                             // declarație incompletă a clasei
class B;                             // declarație incompletă a clasei

class A
{
    int a;
public:
    A(int n)
        {a=n;}
    friend void func(A &, B &);       // funcția func() este declarată prietenă clasei A
};

class B
{
    int b;
public:
    B(int n)
        {b=n;}
    friend void func(A &, B &);       // funcția func() este declarată prietenă clasei B
};

void func( A &ob_a, B &ob_b)         //definirea funcției func(), externă claselor A și B
{
```

```
    cout<<'\\n'<<ob_a.a;
    cout<<'\\n'<<ob_b.b;
}

void main()
{
    A ob1(10);
    B ob2(20);
    func(ob1, ob2);                // apelul funcției func() cu parametri de tip A și
                                   // respectiv B
}
```

Funcția func() se definește în afara claselor A și B, ea fiind o funcție independentă.

```
// Situația 4

#include <iostream.h>
class A;                // declarație incompletă a clasei A
class B                // declarația clasei B
{
    int b;
public:
    B(int n)
        {b=n;}
    void func(A &);
};

class A
{
    int a;
public:
    A(int n)
        {a=n;}
    friend class B;      // clasa B se declară prietenă clasei A
};

void B::func( A &ob_a)
{
    cout<<'\\n'<<ob_a.a;
    cout<<'\\n'<<b;
}

void main()
{
    A ob1(10);
    B ob2(20);
    ob2.func(ob1); }    // apelul funcției func() membră a clasei B
```

Declarația friend din clasa A trebuie să găsească clasa B declarată. Declarația incompletă a clasei A este necesară pentru a se putea specifica parametrii de tip A în funcțiile membre ale clasei B.

Utilizarea funcțiilor prietene încalcă principiul încapsulării, dându-se acces și altor funcții decât celor membre la membrii private ai unei clase. Pot apare probleme și atunci când compilarea funcției prietene se face independent de compilarea funcțiilor membre, prin omiterea accidentală a acestora. De asemenea, dacă funcția este membră altei clase, apar interdependențe între clase.

În ciuda acestor inconveniente, funcțiile prietene oferă anumite facilități, ele oferind o altă soluție de acces controlat către membrii unor clase.

### **Exerciții:**

1. Se definește clasa:

```
class dreptunghi
{ int x1, y1, x2, y2;
  static int cnt;                // membru static ce reprezintă un contor pentru obiectele
                                // clasei create
  void normalizare();           // valorile coordonatelor sunt încadrate în intervalul
                                // [1, 80], cele orizontale și [1, 25] cele verticale
public:
  dreptunghi(int=1, int=1, int=80, int=25);    // coordonatele sunt normalizate
  ~dreptunghi();
  void afisare();                // afișarea se face în mod text, prin marcarea laturilor
                                // dreptunghiului cu un caracter oarecare afișat în mod repetat
  static int nr_dreptunghiuri(); // afișează numărul obiectelor dreptunghi existente la un
                                // moment
  void deplasare(int, int);      // realizează translatarea dreptunghiului; coordonatele
                                // sunt normalizate
};
```

Să se definească o funcție prietenă clasei dreptunghi, aria(), care să returneze aria corespunzătoare dreptunghiului.

În funcția main() se declară obiecte de tip dreptunghi. Se afișează proprietățile obiectelor, ariile corespunzătoare, numărul de instanțieri.

2. Să se definească un tip de dată class data\_calendaristică, un tip de dată class persoana pentru preluarea datelor unei persoane (pentru data nașterii se va folosi un membru de tip data\_calendaristică). În funcția main() se va declara un tablou de tip persoana ce va reprezenta o grupă de studenți. Se citesc date de la tastatură și se realizează ordonarea alfabetică a studenților.

## Supradefinirea operatorilor

### Funcția operator

Limbajul C++ permite programatorului definirea diverselor operații cu obiecte ale claselor, folosind simbolurile operatorilor standard.

Operatorii standard sunt deja supradefiniți, ei putând intra în expresii ai căror operanzi sunt de diferite tipuri fundamentale, operația adecvată fiind selectată în mod similar oricăror funcții supradefinite.

Un tip clasă poate conține definirea unui set de operatori specifici asociați, prin supradefinirea operatorilor existenți, utilizând funcții cu numele:

**operator simbol\_operator**

unde:

- operator este cuvânt cheie dedicat supradefinirii operatorilor;
- simbol\_operator poate fi simbolul oricărui operator, mai puțin operatorii: (.), (\*.), (::) și (? :).

Pentru definirea funcțiilor operator se pot folosi două variante de realizare:

- definirea funcțiilor operator cu funcții membre clasei ;
- definirea funcțiilor operator cu funcții prietene clasei.

Prin supradefinirea operatorilor nu se pot modifica:

- pluralitatea operatorilor (operatorii unari nu pot fi supradefiniți ca operatori binari sau invers);
- precedența și asociativitatea operatorilor.

Operatorii care pot fi supradefiniți în C++ sunt prezentați în Tabelul nr. 7.

**Tabelul nr. 7** – Operatorii care pot fi supradefiniți

Clasa de prioritate	Tip	Operatori	Asociativitate
1	Binar	() [] ->	de la stânga la dreapta
2	Unar	! ~ + - ++ -- & * (tip) new delete	de la dreapta la stânga
3	Binar	->*	de la stânga la dreapta
4	Binar	* / %	de la stânga la dreapta
5	Binar	+ -	de la stânga la dreapta
6	Binar	<< >>	de la stânga la dreapta
7	Binar	< <= > >=	de la stânga la dreapta
8	Binar	= = !=	de la stânga la dreapta
9	Binar	&	de la stânga la dreapta
10	Binar	^	de la stânga la dreapta
11	Binar		de la stânga la dreapta
12	Binar	&&	de la stânga la dreapta
13	Binar		de la stânga la dreapta
14	Binar	= *= /= %= += -= &= ^=  = <<= >>=	de la dreapta la stânga
15	Binar	, (operator virgulă)	de la stânga la dreapta

Funcția operator trebuie să aibă cel puțin un parametru, implicit sau explicit de tipul clasă căruia îi este asociat operatorul. Pentru tipurile standard operatorii își păstrează definirea.

Operatorii =, [], (), -> pot fi definiți doar cu funcții membre nestatice ale clasei.

Programatorul are deplină libertate în modul în care definește noua operație, dar în general pentru a da o bună lizibilitate programului, se recomandă ca noua operație să fie asemănătoare semnificației originale.

## Supradefinirea operatorilor folosind funcții membre clasei

Se consideră clasa complex definită pentru reprezentarea numerelor complexe.

```
class complex
{
    double re, im;
```

```
public:
    complex(double r=0, double i=0)
        { re=r; im=i; }
    void afisare()
        { cout<<"\nre="<<re<<"\tim="<<im; }
}
```

Se dorește definirea unui operator binar + care să realizeze însumarea a doi operanzi de tip complex, rezultatul returnat fiind un obiect complex cu partea reală sumă a părților reale a celor doi operanzi, iar partea imaginară sumă a părților imaginare a celor doi operanzi.

O funcție membră primește implicit adresa obiectului pentru care este apelată, acesta constituind unul dintre parametrii, deci funcția operator + va trebui să aibă un singur parametru explicit. Prototipul funcției va fi:

complex operator+( complex );

Expresia  $x+y$ , unde  $x$  și  $y$  sunt obiecte de tip complex, este echivalentă cu un apel de forma:

$x.operator+(y)$

```
class complex
{
    double re, im;
public:
    complex(double r=0, double i=0)
        { re=r; im=i; }
    void afisare()
        { cout<<"\nre="<<re<<"\tim="<<im; }
    complex operator+(complex);
};

complex complex::operator+(complex c)
{
    complex aux;
    aux.re=re+c.re;
    aux.im=im+c.im;
    return aux;
}

void main()
{
    complex c1(1, 2), c2(3, 4), c3, c4;
    c1.afisare();
    c2.afisare();
}
```

```
c3=c1+c2;  
c3.afisare();  
c4=c1+c2+c3;  
c4.afisare();  
}
```

Programul va afișa:

```
re=1  im=2  
re=3  im=4  
re=4  im=6  
re=8  im=12
```

Expresia  $c3=c1+c2$  este echivalentă cu apelul:

$c3=c1.operator+(c2);$

Pentru evaluarea expresiei  $c4=c1+c2+c3$ , compilatorul creează un obiect temporar de tip complex care preia un rezultat parțial, astfel că expresia este echivalentă cu secvența:

$temp=c1.operator+(c2);$   
 $c3=temp.operator+(c3);$

Se observă că, în cazul definirii funcției `operator` ca funcție membră a clasei, primul operand este obligatoriu de tipul clasei respective.

## Supradefinirea operatorilor folosind funcții prietene clasei

În cazul definirii funcției `operator +` din exemplul anterior cu ajutorul unei funcții prietene, funcția fiind externă clasei, este necesar să preia doi parametri de tip complex, prototipul funcției fiind:

$complex operator+(complex, complex);$

Expresia  $x+y$ , unde  $x$  și  $y$  sunt două obiecte de tip complex, este echivalentă cu un apel al funcției `operator+` de forma:

$operator+(x, y);$

```
class complex
{
    double re, im;
public:
    complex(double r=0, double i=0)
        { re=r; im=i; }
    void afisare()
        { cout<<"\nre="<<re<<"\tim="<<im; }
    friend complex operator+(complex, complex); // funcția operator+() este declarată
                                                // funcție prietenă clasei complex
};

complex operator+(complex a, complex b) // funcția operator+() este funcție externă
                                        // clasei complex
{
    complex c;
    c.re=a.re+b.re;
    c.im=a.im+b.im;
    return c;
}

void main()
{
    complex c1(1, 2), c2(3, 4), c3, c4;
    c1.afisare();
    c2.afisare();
    c3=c1+c2;
    c3.afisare();
    c4=c1+c2+c3;
    c4.afisare();
}
```

Programul va afișa:

```
re=1   im=2
re=3   im=4
re=4   im=6
re=8   im=12
```

Expresia  $c3=c1+c2$  este echivalentă cu apelul:

$c3=operator+(c1, c2);$

La evaluarea expresiei  $c4=c1+c2+c3$  se ține seama de asociativitatea operatorului + de la stânga la dreapta, deci  $c1+c2+c3=(c1+c2)+c3$ .

Apelul funcției operator+ se face astfel:



c3=operator+(operator+(c1, c2), c3);

## Supradefinirea operatorilor unari

Operatorii unari pot fi supradefiniți atât cu funcții membre clasei, cât și cu funcții prietene clasei. Expresia “op x”, unde op este simbolul operatorului supradefinit, este echivalentă cu:

x.operator op();	// pentru definire cu funcție membră
sau	
operator op(x)	// pentru definire cu funcție prietenă

```
#include <iostream.h>

class pozitie {
    int x, y;                                // coordonatele ce descriu poziția
public:
    pozitie (int abs=0, int ord=0)           //constructor cu parametri implicați
    {
        x=abs;
        y=ord;
    }
    void afisare()
    {
        cout<<"x="<<x;
        cout<<", y="<<y<<"\n";
    }
    pozitie operator ++()                    // supradefinire operator ++
    {
        x++; y++;
        cout<<"Op++";
        return pozitie(x, y);
    }
    pozitie operator --()                    // supradefinire operator --
    {
        x--; y--;
        cout<<"Op--";
        return pozitie(x, y);
    }
};
```

```
void main()
{
    pozitie p(1, 1), r;
    r=p++; // expresie echivalentă cu r=p.operator++();
    r.afisare();
    r=-p; // expresie echivalentă cu r=p.operator—(); - diferența
          // dintre pre și post fixarea operatorului nu mai este
          // valabilă

    r.afisare();
    r=++p;
    r.afisare();
    r=p--;
    r.afisare();
}
```

## Supradefinirea operatorului de atribuire (=)

În lipsa supradefinirii operatorului de atribuire pentru operanzi clasă, compilatorul oferă o semnificație predefinită a acestuia, atribuirea făcându-se membru cu membru, în mod asemănător creării constructorului de copiere implicit. Probleme similare utilizării constructorului de copiere apar și la utilizarea operatorului de atribuire.

Operatorul de atribuire poate fi supradefinit doar prin funcții membre clasei respective, nu și prin funcții prietene.

Se consideră clasa complex definită anterior.

```
complex c1(5, 6), c2;
c2=c1;
c2.afisare();
```

Expresia `c2=c1` are ca efect atribuirea valorilor membru cu membru, deci este echivalentă cu atribuirile:

```
c2.re=c1.re;
c2.im=c1.im;
```

Vom analiza în continuare o clasă “vector” destinată creării de tablouri de componente de tip double, având dimensiuni diferite.

```
#include <iostream.h>

class vector
{
    int grad; // preia ca valoare dimensiunea tabloului
```

```

        double* comp;                // preia ca valoare adresa la care se creează tabloul de
                                     // componente
public:
    vector(int gr=10);                // constructor cu parametru implicit
    ~vector();                        // destructor
    void afisare(); };                // funcție de afișare

vector::vector(int gr)
{ cout<<"\nConstructor vector\n";
  grad=gr;
  comp=new double[grad];
  cout<<"\nVectorul are "<<grad<<" componente, introduceti valorile:"
  for(int i=0; i<grad; i++)
      cin>>comp[i];
}

vector::~~vector()
{ delete comp; }

void vector::afisare()
{
    cout<<"\n\nAdresa obiectului este: this="<<this;
    cout<<"\nVectorul are "<< grad << "componente";
    cout<<"\nAdresa membrului comp este: "<<&comp;
    cout<<"\tsi contine valoarea: "<<comp;
    cout<<"\nComponentele vectorului sunt:\n";
    for(int i=0; i<grad; i++)
        cout<<comp[i]<<" ";
}

void main(){
    vector v1(5), v2(3);
    v1.afisare();
    v2.afisare();
    v2=v1;
    v2.afisare();
}

```

Execuția programului are ca rezultat:

```

Constructor vector
Vectorul are 5 componente, introduceti valorile:
1.5                                     // valorile sunt introduse de la tastatură
2.3
5.1
6
9.2

```

Constructor vector

Vectorul are 3 componente, introduceti valorile:

7.1 // valorile sunt introduse de la tastatură

8.22

9.5

Adresa obiectului este: this=0xfff2

Vectorul are 5 componente

Adresa membrului comp este : 0xfff4 si contine valoarea 0x130a

Componentele vectorului sunt:

1,5 2.3 5.1 6 9.2

Adresa obiectului este: this=0xffee

Vectorul are 3 componente

Adresa membrului comp este : 0xfff0 si contine valoarea 0x1336

Componentele vectorului sunt:

7.1 8.22 9.5

Adresa obiectului este: this=0xffee

Vectorul are 5 componente

Adresa membrului comp este : 0xfff0 si contine valoarea 0x1330a

Componentele vectorului sunt:

1.5 2.3 5.1 6 9.2

Se observă că efectul expresiei `v2=v1` este atribuirea valorilor membru cu membru, deci expresia este echivalentă cu:

```
v2.grad=v1.grad;  
v2.comp=v1.comp;
```

Ca urmare, zona de memorie alocată dinamic pentru membrul `v1.comp` prin constructorul obiectului `v1`, va fi referită în continuare și prin `v2.comp`. Dealocarea memoriei prin apelul destructorului pentru cele două obiecte, `v1`, `v2`, se va adresa aceleiași zone de memorie, în schimb, zona alocată inițial de vectorul `v2` va rămâne în continuare alocată.

Având în vedere aceste aspecte, rezultă necesitatea supradefinirii operatorului de atribuire.

```
#include <iostream.h>
```

```
class vector
```

```
{  
    int grad;  
    double* comp;
```

```
public:
```

```
    vector(int gr=10);
```

```
    ~vector();
```

```

        vector(vector&);                // constructor de copiere
        void afisare();
        void operator=(vector &);      // supradefinire operator de atribuire
    };

vector::vector(int gr)
{ cout<<"\nConstructor vector\n";
  grad=gr;
  comp=new double[grad];
  cout<<"\nVectorul are "<<grad <<"componente, introduceti valorile:\n";
  for(int i=0; i<grad; i++)
      cin>>comp[i];
}

vector::~vector()
{ cout<<"\nDestructor obiect cu adresa: "<<this;
  delete comp;
}

vector::vector(vector &v1)
{
  grad=v1.grad;
  comp=new double[grad];
  for(int i=0; i<grad; i++)
      comp[i]=v1.comp[i];
}

void vector::afisare()
{
  cout<<"\n\nAdresa obiectului este: this="<<this;
  cout<<"\nVectorul are "<<grad<<"componente";
  cout<<"\nAdresa membrului comp este: "<<&comp;
  cout<<"\tsi contine valoarea: "<<comp;
  cout<<"\nComponentele vectorului sunt:\n";
  for(int i=0; i<grad; i++)
      cout<<comp[i]<<" ";
}

void vector::operator=(vector &v)
{ if (this!=&v)                // este necesară verificarea apariției unei situații ob=ob
                                // (ob este de tip vector)
    { delete comp;              // dealocarea memoriei corespunzătoare componentelor
                                // inițiale

      grad=v.grad;
      comp=new double[grad];    //se face o nouă alocare pentru un număr de componente
                                // ce poate fi diferit de cel inițial

      for(int i=0; i<grad; i++)
          comp[i]=v.comp[i];   }
}

```

```
void main()
{
    vector v1(5), v2(3);
    v1.afisare();
    v2.afisare();
    v2=v1;
    v2.afisare();
}
```

Execuția programului are un rezultat asemănător programului anterior, dar se poate observa că, în urma atribuirii, membrii comp ai celor doi vectori v1.comp și v2.comp nu mai pointează către aceeași zonă de memorie, ci v2.comp a fost realocat, de data aceasta rezervându-se memorie pentru 5 componente în care s-au copiat valorile corespunzătoare vectorului v1. Operațiile care se efectuează în funcția operator=() sunt:

- eliberarea tabloului de componente al vectorului v2;
- alocarea unui nou tablou de componente de dimensiune egală cu a vectorului v1;
- copierea elementelor tabloului de componente v1 în tabloul de componente v2.

Execuția programului are ca efect afișarea mesajelor:

Constructor vector

Vectorul are 5 componente, introduceți valorile:

```
1.5 // valorile sunt introduse de la tastatură
2.3
5.1
6
9.2
```

Constructor vector

Vectorul are 3 componente, introduceți valorile:

```
7.1 // valorile sunt introduse de la tastatură
8.22
9.5
```

Adresa obiectului este: this=0xfff2

Vectorul are 5 componente

Adresa membrului comp este : 0xfff4 si contine valoarea 0x132a

Componentele vectorului sunt:

1.5 2.3 5.1 6 9.2

Adresa obiectului este: this=0xffee

Vectorul are 3 componente

Adresa membrului comp este : 0xffff0 si contine valoarea 0x1356

Componentele vectorului sunt:

7.1 8.22 9.5

Adresa obiectului este: this=0xffee

Vectorul are 5 componente

Adresa membrului comp este : 0xffff0 si contine valoarea 0x1356

Componentele vectorului sunt:

1,5 2.3 5.1 6 9.2

Atribuirea  $v2=v1$  este echivalentă cu apelul funcției `operator=()` sub forma:

`v2.operator=(v1)`

În definirea funcției `operator=()` s-a specificat ca parametru o referință de vector pentru a crește viteza de execuție și a reduce necesarul de memorie, lucru ce poate deveni important atunci când obiectele transferate prin parametru au dimensiuni semnificative.

În cazul în care se include o expresie de forma  $a=b=c$ , rezultatul este un mesaj de eroare, funcția `operator=()` nereturnând nici o valoare. Pentru a putea folosi atribuiri multiple, funcția trebuie să returneze tipul clasă respectiv, sau referința acelui tip. Prototipul funcției `operator=()` va fi:

`vector& operator=(vector&);`

și funcția va fi definită astfel:

```
vector & vector::operator=(vector &v)
{
    if (this!=&v)
    { delete comp;
      grad=v.grad;
      comp=new double[grad];
      for(int i=0;i<grad;i++)
        comp[i]=v.comp[i];
    }
    return *this;
}
```

Expresia  $v1=v2=v3$  este echivalentă cu:

`v1.operator=(v2.operator=(v3));`

Asociativitatea operatorului de atribuire este de la dreapta la stânga. Rezultatul returnat este chiar obiectul implicit al funcției.

## Supradefinirea operatorului []

Operatorul de indexare [ ] este un operator binar. El are forma generală:

**expresie1[expresie2]**

Funcția **operator[]** se definește ca funcție membră a unei clase (ea nu poate fi definită ca funcție prietenă).

Dacă pentru o clasă A s-a definit funcția operator[](), expresia a[n], unde a este obiect al clasei A, este echivalentă cu :

a.operator[](n)

Clasa vector definită anterior se poate completa cu funcția operator[]() având prototipul:

double& operator[](int);

care, va fi definită astfel:

```
double& vector::operator[](int n)
{ return comp[n]; }
```

Pentru ca operația să poată fi folosită la înscrierea valorilor pentru elementele respective, este necesar ca rezultatul returnat să fie referința elementului cu numărul n.

Funcția main() se poate completa cu secvența:

```
v2[0]=99.9
for (int i=0; i<5; i++)
    cout<<"\ncomp["<<i<<"]="<<v2[i];
```

Dacă elementele componente ale vectorului v2 aveau valorile 1.5, 2.3, 5.1, 6, 9.2, la execuția programului, corespunzător acestei secvențe se va afișa:

```
comp[0]= 99.9      // comp[0] și-a modificat valoarea prin atribuirea v2[0]=99.9
comp[1]= 2.3
comp[2]= 5.1
```



```
comp[3]= 6
comp[4]= 9.2
```

În exemplul următor se declară clasa persoana care preia datele unei persoane (nume și vârstă) și o clasă exemplu\_index care are ca membru un tablou de dimensiune N cu elemente de tip persoana (lista). Se definesc trei criterii de localizare a unui element al listei (după nume, după vârstă și după index) supradefinind funcția operator[]().

```
#include <iostream.h>
#include <string.h>

#define N 10

class persoana
{
    char nume[20];
    double varsta;
    friend exemplu_index;           // clasa exemplu_index este declarată prietenă pentru ca
                                    // funcțiile membre acesteia să aibă acces la membrii
                                    // clasei persoana

public:
    persoana()                     // constructor implicit
    { *nume='\0';
      varsta=0;
    }
    void date_persoana(char * p, double v) // funcție membră prin care se transmit
                                            // valori pentru membrii obiectului
    { strncpy(nume, p, 19);
      nume[19]='\0';
      varsta=v;
    }
    void afisare()
    { if (this!=NULL)
      cout<<"\nNume: "<<nume<<" varsta: "<<varsta<<" ani";
      else
      cout<<"\nElement inexistent in lista";  }
};

class exemplu_index
{
    persoana lista[N];             // membrul lista este un tablou de N elemente de
                                    // tip persoana

public:
    void citire_date(void);        // citirea datelor de la tastatură pentru cele N
                                    // persoane din lista
}
```

```
persoana * operator[](char *);           // funcție operator[]() care localizează elementul
                                           // după nume
persoana * operator[](double);           // funcție operator[]() care localizează elementul
                                           // după vârstă
persoana * operator[](int);              // funcție operator[]() care localizează elementul
                                           // după index
};

void exemplu_index::citire_date()
{
    char n[20];
    double d;
    for(int i=0; i<N; i++)
        { cout<<"\nTastati numele si varsta: ";
          cin>>n>>d;
          lista[i].date_persoana(n, d);
        }
}

persoana * exemplu_index::operator[](char * string )
{
    for (int i=0; i<N; i++)
        if (strcmp(lista[i].nume, string)==0)
            return &lista[i];
    return NULL;
}

persoana * exemplu_index::operator[](double varsta)
{
    for (int i=0; i<N; i++)
        if (lista[i].varsta==varsta)
            return &lista[i];
    return NULL;
}

persoana * exemplu_index::operator[](int i )
{
    if (i<N)
        return &lista[i];
    return NULL;
}

void main(void)
{
    exemplu_index lista_pers;           // se declară un obiect exemplu_index
    lista_pers.citire_date();            // se citesc datele persoanelor din lista
    lista_pers["Adrian"]->afisare();     // se caută în lista elementul cu numele="Adrian"
    lista_pers[(double)22]->afisare();   // se caută în lista elementul cu varsta=22;
```

---

```

// conversia este necesară pentru a impune apelul
// funcției cu căutare după vârstă, altfel, valoarea
// parametrului fiind de tip int, se apelează funcția ce
// face localizarea după index
lista_pers[2]->afisare();
}

```

În funcția de afișare s-a introdus verificarea adresei obiectului persoana. Funcțiile operator[]() returnează adresa obiectelor persoana dacă localizarea elementului s-a putut face, în caz contrar returnează valoarea 0 (NULL).

## Supradefinirea operatorilor new și delete

Operatorii **new** și **delete** sunt supradefiniți astfel încât se admite orice tip de dată, inclusiv tipul clasă. Ei pot fi supraîncărcați pentru clasele definite, oferind operații specializate de alocare, respectiv eliberare dinamică a memoriei.

Funcțiile operator new și delete pot fi definite ca funcții membre ale clasei, nu și ca funcții prietene acestuia. Acestea sunt funcții implicit statice, chiar dacă nu sunt declarate static explicit.

Funcția operator new trebuie să aibă un parametru de tipul `size_t` care precizează dimensiunea în octeți a obiectului alocat și să returneze un pointer void (`void*`) care să conțină adresa zonei de memorie alocată obiectului:

**`void * operator new(size_t);`**

Tipul **`size_t`** este declarat în `stdlib.h` ( `alloc.h`, `stddef.h`, `mem.h`, etc.). Parametrul `size_t`, chiar dacă este specificat de programator, calculul dimensiunii obiectului revine compilatorului. Din acest motiv, la apelul funcției nu este obligatoriu să se specifice argumentul, deci nu se modifică sintaxa de utilizare.

Operatorul new supradefinit păstrează proprietatea de a apela constructorul clasei după alocarea dinamică a memoriei necesare obiectului.

Funcția **operator delete** trebuie să primească un parametru pointer de tipul clasei asociate sau void care, la apel, conține adresa obiectului care se elimină. Funcția nu returnează rezultat. Funcția operator delete poate să aibă un al doilea parametru, opțional, de tip `size_t`.

**`void operator delete (void *)`**

Operatorul delete supradefinit păstrează proprietatea de a apela destructorul clasei înainte de a elibera zona de memorie a obiectului dinamic.

Dacă sunt supradefiniți operatorii new și delete, definițiile standard pot fi utilizate în continuare folosind operatorul de rezoluție (::).

```
#include <iostream.h>
#include <stddef.h>

class vector
{
    int grad;
    double* comp;
    static int cnt;                // membru static, utilizat ca și contor pentru
                                   // obiectele vector dinamice

public:
    vector(int gr=10);             // constructor cu parametru implicit
    ~vector();                     // destructor
    void afisare();
    void * operator new(size_t dim); // prototip funcție operator new()
    void operator delete(void *);    // prototip funcție operator delete()
};

int vector::cnt=0;

vector::vector(int gr)             // definire constructor cu parametru implicit
{
    cout<<"\nConstructor vector\n";
    grad=gr;
    comp=new double[grad];
    cout<<"\nVectorul are "<<grad <<"componente, introduceti valorile:\n";
    for(int i=0; i<gr; i++)
        cin>>comp[i];
}

vector::~~vector()                 // definire destructor
{
    cout<<"\nDestructor obiect cu adresa: "<<this;
    delete comp;
}

void * vector::operator new(size_t dim) // supradefinirea funcției operator new()
{
    cnt++;                          // la crearea unui obiect dinamic de tip
                                   // vector, contorul se incrementează

    cout<<"\ndim="<<dim<<"\nnew: cnt="<< cnt;
    return new char[dim];           // valoarea returnată este adresa memoriei
                                   // alocate prin operatorul new predefinit
}

void vector::operator delete(void * p) // supradefinirea funcției operator delete()
{

```

```

        cout<<"\ndelete: cnt= "<<cnt;
        cnt--;                                // la distrugerea unui obiect dinamic vector,
                                              // contorul se decrementează

        delete p;
    }

void vector::afisare()
{
    cout<<"\nVectorul are "<<grad<<"componente";
    cout<<"\nComponentele vectorului sunt:\n";
    for(int i=0;i<grad;i++)
        cout<<comp[i]<<" ";
}

void main()
{
    vector *pv1, *pv2, *pv3;                // se declară pointeri la tipul vector
    pv1=new vector(5);                      // se alocă dinamic memorie pentru un obiect vector;
                                              // operator new apelează constructorul cu parametru
                                              // implicit, valoarea parametrului fiind 5
    pv2=new vector(*pv1);                   // se alocă dinamic memorie pentru un obiect vector,
                                              // copie a celui de la adresa depusă în pv1, ca urmare se
                                              // apelează constructorul de copiere
    pv3=new vector(2);                      // alocarea de memorie se face cu operatorul new
                                              // predefinit

    pv1->afisare();
    pv2->afisare();
    pv3->afisare();
    delete pv1;                            // se elimină obiectul pv1, apelându-se operatorul delete
                                              // definit de utilizator
    delete pv2;                            // se elimină obiectul pv1, apelându-se operatorul delete
                                              // definit de utilizator

    ::delete pv3;                          // se elimină obiectul pv1, apelându-se operatorul delete
                                              // predefinit
}

```

Operatorii new și delete pot fi supradefiniți folosind funcții independente, cu prototip similar celor anterioare. În această situație, operatorii new și delete vor fi apelați pentru toate tipurile de date, inclusiv cele standard. Operatorii predefiniți nu mai pot fi folosiți, deci programatorul este cel care trebuie să controleze gestionarea alocării dinamice a memoriei .

### Exerciții:

1. Se definește clasa de matrici pătratice:

```
class matrice
{
    int dim;
    double * m ;
public:
    matrice();
    ~matrice();
    void citeste();
    void afiseaza();
};
```

Să se completeze clasa cu operatorul unar - (opusul matricei) și cu operatorii binari + (suma), \* (produsul) definiți prin funcții membre sau prietene și operatorul de atribuire. Să se definească o funcție prietenă care să returneze transpusa matricii.

2. Fie clasa :  $Q = \{ p/q \mid p, q \in \mathbb{Z}, q \neq 0 \}$  definită mai jos:

```
class rational
{
    int p, q;
    void simplifica();           // aduce membrii p și q în formă ireductibilă
public:
    rational(int a=0, b=1);      // se verifica b≠0 și se aduc p și q în formă
                                // ireductibilă
    ~rational();
    void afiseaza();
};
```

Să se completeze clasa cu operatorii unari și binari corespunzători operațiilor specifice numerelor raționale, folosind atât funcții membre, cât și funcții prietene clasei, membrii p și q fiind aduși în formă ireductibilă după fiecare modificare a lor.

# Conversii de tip definite de utilizator

## Procedee de definire a conversiilor

În timpul execuției unui program, compilatorul efectuează conversii implicite, în conformitate cu regulile deja cunoscute, în următoarele situații:

- la evaluarea unei expresii, pentru fiecare operator, dacă operandii sunt de tipuri diferite, se efectuează conversia către operandul de tip cu domeniu de valori mai mare a celui cu domeniu de valori mai mic;
- la atribuire, se face conversia valorii atribuite către tipul operandului care primește valoarea;
- la transferul parametrilor, se face conversia parametrilor efectivi către tipul de dată specificat în prototipul funcției.

În limbajul C++, se pot defini conversii explicite de tip folosind operatorul “cast”.

Pentru clase se pot defini reguli de conversie care se adaugă conversiilor standard și sunt apelate de compilator în toate situațiile specificate mai sus. Acestea sunt utilizate cu unele restricții:

- într-un lanț de conversii, compilatorul admite o singură conversie definită de utilizator;
- conversiile definite de utilizator se folosesc doar dacă nu există alte soluții (de exemplu, la atribuire, întâi se verifică existența supradefinirii operatorului de atribuire și numai în absența acesteia se va aplica conversia definită de utilizator).

Există două modalități de definire a conversiilor. O primă variantă constă în supraîncărcarea operatorului unar cast folosind o funcție membră a clasei.

De exemplu, pentru tipul complex se pot defini conversii la tipul double și la tipul poziție:

```
complex::operator double(),
```

respectiv

complex::operator pozitie()

A doua variantă constă în definirea constructorului cu un singur parametru de tipul din care se face conversia. De exemplu:

complex(double)  
complex(pozitie)

**Observații:**

- supradefinirea operatorului cast nu poate realiza conversii dintr-un tip fundamental într-un tip clasă;
- supradefinirea constructorului nu permite conversia unui tip clasă într-un tip fundamental.

## Supradefinirea operatorului cast folosind funcții membre clasei

Operatorul cast este un operator unar. Supradefinirea operatorului cast se poate face numai cu funcții membre clasei din care se face conversia. În declarația și definiția funcției nu trebuie specificat tipul rezultatului, acesta fiind specificat, implicit, prin numele operatorului. Funcția nu are nici parametru, ea fiind funcție membră a clasei primește adresa obiectului, ca orice funcție membră.

Se definește clasa complex care va conține funcția membră operator float() care va face conversia unui obiect complex în dată de tip float, valoarea returnată fiind partea reală (membrul re) a numărului complex.

```
#include <iostream.h>

class complex
{
    float re, im;
public:
    complex(float r=0, float i=0)
    {
        cout<<"\nConstructor: ";
        re=r; im=i;
        afisare();
    }
    complex(complex &c)
    {
        cout<<"\nConstructor de copiere: ";
```



```
        re=c.re; im=c.im;
        afisare();
    }
    operator float()
    {
        cout<<"\nApel float-valoare returnata:"<<re;
        return re;
    }

    void afisare()
    {   cout<<"re="<<re<<"\tim="<<im; }
};

void fct(float r)
{
    cout<<"\nApel functie: valoare float="<<r;
}

void main()
{
    complex c1(2, 2), c2(5, 5);
    float r1, r2;

    // 1. Conversie la atribuire
    cout<<"\n1.1 conversie explicita la atribuire";
    r1=(float)c1;
    cout<<"\tr1="<<r1;

    cout<<"\n1.2. conversie implicita la atribuire";
    r2=c2;
    cout<<"\tr2="<<r2;

    // 2. Conversii la transfer de parametri
    cout<<"\n2.1 apel al functie fara conversie a parametrului";
    fct(r1);

    cout<<"\n2.2 apel al functie cu conversie explicita a parametrului ";
    fct((float)c1);

    cout<<"\n2.3 apel al functiei cu conversie implicita a parametrului";
    fct(c2);

    // 3. Conversii la evaluarea expresiilor
    cout<<"\n3.1 evaluarea expresiilor - conversie explicita a operandului";
    r1=(float)c1+r1;
    cout<<"\tr1="<<r1;

    cout<<"\n3.2.a evaluarea expresiilor - conversie implicita a unui operand";
```

```
    r1=c1+r1;
    cout<<"\tr1="<<r1;

    cout<<"\n3.2.b evaluarea expresiilor - conversie implicita a ambilor operanzi";
    r1=c1+c2;
    cout<<"\tr1="<<r1;

    cout<<"\n3.2.c evaluarea expresiilor - conversie implicita complex->float\
și double->float ";
    r1=c1+2.5;
    cout<<"\tr1="<<r1;
}
```

În cazul atribuirii, exemplele 1.1 și 1.2, se observă că, indiferent că se realizează o conversie implicită sau explicită, se apelează operatorul de conversie pentru membrul drept.

În cazul apelurilor funcției `fct()`, se observă că în situațiile care presupun conversii de tip, întâi se efectuează conversia de la `complex` către `float`, valoarea rezultată fiind preluată de parametru și apoi se apelează funcția. Astfel se explică faptul că nu se apelează constructorul de copiere pentru preluarea valorii de către parametrul funcției.

În cazul expresiei `c1+r1`, compilatorul verifică inițial existența supradefinirii operatorului `+` pentru operanzi, unul `complex`, celălalt `float`. În lipsa acestuia, se caută definirea unei conversii care să permită operația între doi operanzi de același tip.

Pentru expresia `c1+c2`, în mod similar situației anterioare, se va face conversia ambilor operanzi către tipul obiectului care preia valoarea, deci amândoi operanzii `complex` se convertesc la tipul `float`.

În expresia `c1+2.5`, constanta `2.5` este memorată în format `double`, deci `c1` va cunoaște conversia `complex->float->double`. În această succesiune, doar o conversie este definită de utilizator, deci ea va fi acceptată de compilator.

Dacă în timpul efectuării conversiilor compilatorul sesizează existența mai multor variante de conversii, ambiguitatea va fi semnalată cu mesaj de eroare.

## **Supradefinirea operatorului cast folosind funcții prietene clasei**

Operator `cast` poate fi utilizat și pentru definirea unor conversii a unui tip clasă în alt tip clasă. În această situație, funcțiile operator `cast` folosite sunt funcții prietene acesteia.

Se definesc clasele `complex` și `poziție`, definindu-se totodată operatorul de

conversie complex -> pozitie.

```
#include <iostream.h>

class complex;           //declarație incompletă a clasei complex
class pozitie
{
    int x, y ;
public:
    pozitie(int abs=0, int ord=0)           //constructor pozitie cu parametri impliciți
    { x=abs;
      y=ord;
      cout<<"\nConstructor pozitie ";
      afisare();
    }
    void afisare()
    { cout<<"x="<<x;
      cout<<", y="<<y;
    }
    operator complex();           // declarare conversie pozitie->complex
};

class complex {
    float re, im;
public:
    complex(float r=0, float i=0)           //constructor complex
    { re=r; im=i; }
    void afisare()
    { cout<<"(re="<<re;
      cout<<", im="<<im<<")";
    }
    friend complex operator +(complex, complex); // supradefinire operator +() binar pentru
                                                    // clasa complex
    friend pozitie::operator complex();         // declararea funcției operator complex() membră
                                                    // a clasei pozitie, prietenă clasei complex
};

pozitie::operator complex()
{ cout<<"\nOperator pozitie->complex:";
  complex c;
  c.re=x;
  c.im=y;
  cout<<"("<<x<<","<<y<<")->";
  c.afisare();
  return c;
}
```

```
complex operator +(complex a, complex b)
{ cout<<"\nOperator + complex:";
  a.afisare();
  cout<<" + ";
  b.afisare();
  cout<<"\n";
  return complex(a.re+b.re, a.im+b.im);
}

void main()
{
  complex c1(1.1, 1.1), r;
  pozitie p1(10, 10), p2(20, 20);
  cout<<"\n";

  // conversie complex=complex+pozitie
  r=c1+p2;
  cout<<"r : ";
  r.afisare();
  cout<<"\n\n";

  //conversie complex=pozitie+pozitie
  r=p1+p2;
  cout<<"r : ";
  r.afisare();
  cout<<"\n\n";
}
```

Execuția programului are ca rezultat afișarea:

```
Constructor pozitie: x=10, y=10
Constructor pozitie: x=20, y=20

Operator pozitie->complex: (20, 20)->re=(20, im=20)
Operator + complex : (re=1.1, im=1.1) + (re=20, im=20)
r: (re=21.1, im=21.1)

Operator pozitie->complex: (20, 20)->(re=10, im=20)
Operator pozitie->complex: (10, 10)->(re=10, im=10)
Operator + complex : (re=10, im=10) + (re=20, im=20)
r: (re=30, im=30)
```

Pentru ca în declararea clasei poziție să poată fi inclus operator complex(), a fost necesară întâi declararea incompletă a clasei complex.

Funcția operator complex() trebuie să aibă acces la membrii clasei complex, deci ea a trebuit declarată prietenă acesteia.

## Conversii de tip folosind constructori

### Conversia unei date de tip standard în dată de tip clasă

De exemplu, pentru clasa `complex` definită anterior, datorită faptului că a fost definit un constructor cu parametri implicați, se poate obține conversia `float -> complex` prin specificarea unui singur parametru la crearea obiectului `complex`.

Se consideră exemplul următor:

```
#include <iostream.h>

class complex
{
    float re, im;
public:
    complex(float r=0, float i=0) //constructor cu parametri implicați
    {
        cout<<"\nConstructor : " ;
        re=r; im=i;
        afisare();
    }
    void afisare()
    { cout<<"re="<<re;
      cout<<" , im="<<im;
    }
};

void f(complex c)
{
    cout<<"\nApel functie : ";
    c.afisare();
}

void main()
{
    complex c1(1, 1), c2(2, 2);
    float r1=10, r2=20;

    // conversie explicita float->complex
    c1=complex(r1);
    cout<<endl;
    c1.afisare();

    // conversie implicita float->complex la atribuire
    c2=r2;
```

```
cout<<endl;
c2.afisare();

// conversie implicita float->complex la transfer de parametri
f(r1);

// conversie implicită int->float->complex la atribuire
int i=7;
c2=i;
cout<<endl;
c2.afisare();
}
```

Pe ecran vor apărea mesajele:

```
Constructor: re=1, im=1
Constructor: re=2, im=2
Constructor: re=10, im=10
re=10, im=0
Constructor: re=20, im=0
re=20, im=0
Constructor: re=10, im=0
Apel functie: re=10, im=0
Constructor: re=7, im=0
re=7, im=0
```

Se observă că în toate situațiile s-a creat un obiect temporar de tip complex care preia pentru partea reală valoarea float, iar membrul im preia valoarea implicită 0, apoi se face copierea, membru cu membru, în obiectul destinație.

Constructorul poate fi folosit și pentru alte conversii de tip, de exemplu int -> complex, valoarea int fiind întâi convertită la tipul float și în final are loc conversia float -> complex.

### **Conversia unui tip clasă în alt tip clasă**

Se poate defini conversia unui tip clasă în alt tip clasă, incluzând în declarația clasei către care se face conversia, un constructor cu un parametru dinspre care se face conversia.

În exemplul următor se va defini conversia tipului pozitie în tipul complex.

```
#include <iostream.h>
class pozitie;
class complex
```

```
{
    float re, im;
public:
    complex(float r=0, float i=0)
        { re=r; im=i; }
    complex(pozitie);                // constructor cu parametru pozitie care permite
                                    // conversia pozitie->complex

    void afisare()
        { cout<<"\nComplex: re="<<re;
          cout<<" , im="<<im;
        }
    complex operator+(complex c)
        {
            complex aux;
            aux.re=re+c.re;
            aux.im=im+c.im;
            return aux;
        }
};

class pozitie
{
    int x, y;
public:
    pozitie(int abs=0, int ord=0)
        {x=abs; y=ord;}
    void afisare()
        {cout<<"\nPozitie: x="<<x<<"\ty="<<y;}
    friend complex::complex(pozitie); // constructorul clasei complex se declară
                                        // funcție prietenă
};

complex::complex(pozitie p)
{
    re=p.x; im=p.y;
}

void main()
{
    complex c1, c2;
    pozitie p1(5, 5), p2(7, 7);

    // conversie pozitie->complex explicită
    c1=complex(p1);
    cout<<endl;
    p1.afisare();
    c1.afisare();
}
```

```
// conversie pozitie->complex implicită la atribuire
c2=p2;
cout<<endl;
p2.afisare();
c2.afisare();

// conversii implicite la evaluarea expresiilor
c1=p1+p2;          // conversii pozitie->complex
cout<<endl;
c1.afisare();

c1=p1+5.5;          // pentru p1 se face conversie pozitie->complex
                    // pentru 5.5 au loc conversiile double->float->complex

cout<<endl;
c1.afisare();
}
```

Pentru a putea declara constructorul folosit pentru conversia `pozitie->complex`, este necesară declararea incompletă a clasei `pozitie` înainte de declararea clasei `complex`. Acest constructor trebuie declarat funcție prietenă clasei `pozitie` pentru a avea acces la membrii acesteia.

La evaluarea expresiei `p1+p2`, se observă extinderea, prin conversie, a operatorului `+` supradefinit în clasa `complex`. În lipsa supradefinirii operatorului `+` în clasa `pozitie`, compilatorul va efectua conversia operanzilor `p1` și `p2` la tipul `complex`, rezultatul fiind, de asemenea, de tip `complex`. Având în vedere tipul rezultatului, rezultă că acesta nu poate fi atribuit unui obiect de tip `pozitie`, deci o expresie de forma `p1=p1+p2` va genera un mesaj de eroare.

În cazul expresiei `p1+5.5`, în lipsa supradefinirii operatorului `+` pentru un operand `pozitie` și celălalt `double`, se va apela, de asemenea, operatorul `+` supradefinit pentru clasa `complex`. Operandul `p1` este convertit la tipul `complex` prin constructorul cu parametru `pozitie`, iar constanta de tip `double` `5.5` este convertită la tipul `float` și apoi, prin constructorul cu parametri implicați, este convertită la tipul `complex`.

**Observații:-** Operatorul cast este un operator unar. El se poate defini cu o funcție membră a clasei, rezultatul returnat fiind obligatoriu de tipul operatorului;

- Supradefinirea operatorului cast nu se poate folosi pentru a realiza conversii dintr-un tip fundamental într-un tip clasă;
- Constructorul unei clase nu permite conversia unui obiect de tip clasă într-un tip fundamental, deoarece funcția constructor returnează implicit un obiect de tipul clasei căreia îi aparține;
- Conversii de la un tip clasă la alt tip clasă se pot realiza atât prin



supradefinirea operatorului cast, cât și prin funcții constructor. În ambele situații este importantă ordinea plasării declarațiilor claselor. Nu este permisă definirea unui tip de conversie prin ambele variante, acest lucru conducând la o ambiguitate semnalată prin mesaj de eroare de compilator.

- În situația în care un operator este supradefinit într-o clasă, la evaluarea expresiilor care conțin operatori de tipuri diferite clasei, compilatorul va extinde funcționarea respectivului operator pentru operanzi de tipuri pentru care nu există supradefinit operatorul, dacă conversiile necesare sunt definite și lanțul de conversii nu necesită mai mult de una definită de utilizator.

### **Exerciții:**

1. Să se definească clasa `cerc` ce descrie un cerc prin rază și coordonatele centrului. Să se definească conversii de la tipuri de date fundamentale la tipul `cerc`, de la tipul `cerc` la tipuri fundamentale. Să se definească conversii de la și către clasa `pozitie` ce conține coordonatele unui punct.
2. Să se definească clasa `vector_pozitie` ce descrie un vector de poziție prin modul și unghiul față de abscisă. Să se supradefinească operatorul binar `+` care să compună doi vectori. Având definită clasa `cerc` de la exercițiul nr. 1, să se definească conversiile `vector_pozitie->cerc` (raza cercului preia valoarea modului, coordonatele centrului fiind (0,0)) și `cerc->vector_pozitie` (modulul vectorului preia valoarea razei, unghiul fiind 0). Să se evidențieze extinderea funcționării operatorului `+` supradefinit asupra obiectelor de tip `cerc`.

### Moștenirea. Clase derivate

Prin procedeul de derivare se urmărește obținerea unor clase noi, numite **clase derivate**, care moștenesc proprietățile unei clase deja definite, numită **clasă de bază**.

Clasele derivate conțin toți membrii clasei de bază, la care se adaugă noi membrii, date și funcții membre.

Clasa de bază nu este afectată de derivare, ea putând fi compilată anterior eventualelor derivări.

Dintr-o clasă de bază se poate deriva o clasă care, la rândul său, să servească drept clasă de bază pentru derivarea altora. Prin această succesiune se obține o **ierarhie de clase**. Pornind de la clase simple și generale, fiecare nivel al ierarhiei adaugă noi proprietăți, obținându-se clase cu un grad sporit de complexitate, devenind din ce în ce mai specializate.

Se pot defini clase derivate care au la bază mai multe clase, înglobând proprietățile tuturor claselor de bază, procedeu ce poartă denumirea de **moștenire multiplă**.

### Declararea clasei derivate

După cum s-a văzut în lucrările de laborator anterioare, sintaxa generală de declarare a unui tip de date class este de forma:

```
class <nume_clasa> <: lista_clase_baza> {<lista_membri>}  
    <lista_variabile>;
```

lista\_clase\_baza cuprinde unul (în cazul unei derivări simple) sau mai multe (în cazul unei derivări multiple) nume de clase, însoțite de câte un specificator de acces. Specificatorul de acces controlează drepturile de acces la membrii clasei de bază. Deci lista claselor de bază are sintaxa următoare:

**specificator\_acces clasa\_baza\_1, specificator acces clasa\_baza\_2,...**

Specificatorii de acces ce se pot utiliza sunt **public** și **private**. Valoarea implicită este **private**.

Atributele de acces la membrii moșteniți de clasa derivată, în funcție de specificatorul de acces folosit, sunt cele prezentate în Tabelul nr. 8.

**Tabelul nr. 8** Atributele de acces la membrii moșteniți

Atributul din clasa de bază	Modificatorul de acces	Accesul moștenit de clasa derivată	Accesul din exterior
private	private	inaccesibil	inaccesibil
protected		private	inaccesibil
public		private	inaccesibil
private	public	inaccesibil	inaccesibil
protected		protected	inaccesibil
public		public	accesibil

Pentru a avea acces la membrii clasei de bază din clasa derivată, este necesar ca aceștia să fi fost declarați **protected** sau **public**.

Pentru a păstra dreptul de acces la membrii clasei de bază, este necesar să se folosească **acces public**.

La stabilirea specificatorilor de acces se au în vedere perspectivele de dezvoltare a ierarhiei de clase, fără a se încălca principiul încapsulării datelor.

Moștenirea este asemănătoare cu procesul de includere a obiectelor în obiecte (procedeu ce poartă denumirea de **compunere**), dar există câteva elemente caracteristice moștenirii:

- codul poate fi comun mai multor clase;
- clasele pot fi extinse, fără a recompila clasele originare;
- funcțiile ce utilizează obiecte din clasa de bază pot utiliza și obiecte din clasele derivate din această clasă.

```
#include <iostream.h>
```

```
class punct  
{
```

```
    float x, y;
```

```
public :
```

```
    punct(float a=0, float b=0)
```

```
    {x=a; y=b;}
```

```
void modific_punct(float dx, float dy)
    {x+=dx; y+=dy;}
void afisare()
    {cout<<"\nPunct: x="<<x<<"\ty="<<y;}
};

class cerc
{
    punct centru;
    float raza;
public:
    cerc(float r=0)
        {raza=r;}
    void afisare()
        {centru.afisare();           // afișarea valorilor corespunzătoare centrului, x și
                                     // respectiv y, se face prin funcția membră clasei punct;
                                     // nu este permis acces direct deoarece membrii x și y
                                     // sunt private

        cout<<"\tRaza="<<raza; }
    void modific_cerc(float dx, float dy, float dr)
        {centru.modific_punct(dx, dy); // modificarea valorilor corespunzătoare
                                     // centrului se face prin funcția membră clasei
                                     // punct; nu este permis acces direct deoarece
                                     // membrii x și y sunt private

        raza+=dr; }
};

void main()
{
    cerc c1;
    c1.modific_cerc(1.1, 2.2, 3.3);
    c1.afisare();
}
```

Clasa cerc include un membru de tip punct, centru. Accesul la membrii privați ai clasei punct se face doar prin funcțiile membre cu acces public ale clasei punct. Pentru a avea acces direct la toți membrii, ar trebui declarați toți cu acces public, ceea ce nu mai permite controlul strict asupra membrilor clasei punct, deci este încălcat principiul încapsulării.

În exemplul următor se va deriva clasa cerc din clasa punct.

```
#include <iostream.h>

class punct
{
```

```

protected:                // se folosește specificatorul de acces protected pentru a permite
                           // accesul la datele membre ale clasei punct din clasele derivate,
                           // dar asigurându-se în continuare protecția față de funcțiile
                           // externe acestora

    float x, y;
public :
    punct(float a=0, float b=0)
        {x=a; y=b;}
    void modific_punct(float dx, float dy)
        {x+=dx; y+=dy;}
};

class cerc : public punct    // în declarație se specifică clasa de bază punct cu acces public
{
    float raza;
public:
    cerc(float r=0)
        {raza=r;}
    void afisare()
        {cout<<"\nCentru cerc: x="<<x<<"\ty="<<y; // accesul la membrii clasei punct se face
                                                    // direct datorită specificatorului
                                                    // protected din clasa punct și a
                                                    // specificatorului public atașat clasei de
                                                    // bază

        cout<<"\tRaza="<<raza;
        }
    void modific_cerc(float dx, float dy, float dr)
        { x+=dx;                                // accesul la membrii clasei punct se face direct
          y+=dy;
          raza+=dr;}
};

void main()
{
    cerc c1;
    c1.modific_cerc(1.1, 2.2, 3.3);
    c1.afisare();
    c1.modific_punct(2.5, 4.6);                // accesul la funcțiile membre clasei punct prin
                                                    // obiecte de tip cerc este permis
    c1.afisare();
}

```

Clasa derivată preia toate proprietățile clasei de bază (membrii x, y, funcțiile punct() și modific\_punct()), la acestea adăugând membrul raza, funcția cerc(), funcția modific\_cerc() și funcția afisare().

Este permisă atribuirea obiectelor derivate unor obiecte ale clasei de bază, deci în funcția main() a exemplului anterior se poate include secvența:

```
punct p1;           // se creează un obiect punct cu x=0 și y=0
p1=c1;              // p1.x preia valoarea c1.x, iar p1.y preia valoarea c1.y
```

Prin operația de atribuire se preiau numai membrii clasei de bază. Regula de compatibilitate se poate extinde și la pointeri și referințe de obiecte.

```
punct * p_p;
cerc * p_c;
...
p_p=p_c;
```

## Constructorii și destructorii pentru clasa derivată

La crearea unui obiect se apelează întotdeauna constructorul clasei respective, iar declarația obiectului trebuie să specifice valorile inițiale cerute de constructor. Distrugerea obiectului este precedată de apelul funcției destructor a clasei respective.

Aceste reguli sunt valabile și în cazul claselor derivate. Pentru crearea unui obiect al unei clase derivate, se creează inițial un obiect al clasei de bază prin apelul constructorului acesteia, apoi se adaugă elementele specifice clasei derivate prin apelul constructorului clasei derivate. Declarația obiectului derivat trebuie să conțină valorile de inițializare, atât pentru elementele specifice, cât și pentru obiectul clasei de bază. Deci la definirea constructorului clasei derivate, este necesar să se specifice care sunt parametrii ce sunt preluați de constructorul clasei de bază. Această specificare se atașează la antetul funcției constructor a clasei derivate.

```
#include <iostream.h>

class punct
{
protected:
    float x, y;
public :
    punct(float a, float b)
    {
        x=a; y=b;
        cout<<"\nConstructor punct: this="<<this;
        cout<<"\tx="<<x<<"\ty="<<y;
    }
}
```

```

    ~punct()
    {
        cout<<"\nDestructor punct: this="<<this;
        cout<<"\tx="<<x<<"\ty="<<y<<endl;
    }
};

class cerc:public punct                // se declară clasa cerc derivată din clasa punct
{
    float raza;
public:
    cerc(float a, float b, float r):punct(a, b)    // în antetul constructorului se specifică efectiv
                                                    // care sunt parametrii transferați constructorului
                                                    // punct
    {
        raza=r;
        cout<<"\nConstructor cerc: this="<<this;
        cout<<"\traza="<<r<<endl;
    }
    ~cerc()
    {
        cout<<"\nDestructor cerc: this="<<this;
        cout<<"\traza="<<raza;
    }
};

void main()
{
    cerc c1(1.1, 2.2, 10), c2(5, 10, 15);
}

```

În urma execuției programului, se va afișa:

Constructor punct: this=0xffea	x=1.1 y=2.2
Constructor cerc: this=0xffea	raza=10
Constructor punct: this=0xffde	x=5 y=10
Constructor cerc: this=0xffde	raza=15
Destructor cerc: this=0xffde	raza=15
Destructor punct: this=0xffde	x=5 y=10
Destructor cerc: this=0xffea	raza=10
Destructor punct: this=0xffea	x=1.1 y=2.2

La definirea constructorului clasei cerc, s-a specificat care dintre parametrii transmiși sunt preluați de constructorul clasei punct.

S-a inclus afișarea de mesaje, astfel încât să se poată urmări ordinea în care se fac apelurile funcțiilor.

Se observă că, la crearea unui obiect cerc, se apelează întâi constructorul punct și apoi constructorul cerc. De asemenea, se observă că adresa obiectului este unică, ea fiind asociată tuturor elementelor care alcătuiesc un obiect cerc, deci ambii constructori, atât punct(), cât și cerc() vor afișa aceeași valoare.

La eliminarea obiectelor cerc, apelul destructorilor se face în ordine inversă, deci întâi se apelează destructorul ~cerc() și apoi ~punct().

În situația în care clasele de bază au definit constructor implicit sau constructor cu parametri implicați, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază. Astfel se explică corectitudinea exemplului prezentat în paragraful anterior.

## Constructorul de copiere pentru o clasă derivată

În ceea ce privește utilizarea constructorului de copiere pentru o clasă derivată, se pot distinge mai multe situații.

Dacă ambele clase, atât clasa derivată cât și clasa de bază, nu au definit constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.

```
#include <iostream.h>

class punct
{
protected:
    float x, y;
public :
    punct(float a, float b)
    {
        x=a; y=b;
        cout<<"\nConstructor punct:";
        cout<<"\tx="<<x<<"\ty="<<y;
    }

    ~punct()
    {
        cout<<"\nDestructor punct: ";
        cout<<"\tx="<<x<<"\ty="<<y<<endl;
    }
};
```



```
class cerc:public punct
{
    float raza;
public:
    cerc(float a, float b, float r):punct(a, b)
    {
        raza=r;
        cout<<"\nConstructor cerc: ";
        cout<<"\traza="<<r<<endl;
    }
    ~cerc()
    {
        cout<<"\nDestructor cerc: ";
        cout<<"\traza="<<raza;
    }
};

void main()
{
    cerc c1(1.1, 2.2, 10);
    cerc c2(c1);
}
```

Programul afișează:

Constructor punct:	x=1.1 y=2.2
Constructor cerc:	raza=10
Destructor cerc:	raza=10
Destructor punct:	x=1.1 y=2.2
Destructor cerc:	raza=10
Destructor punct:	x=1.1 y=2.2

Crearea obiectului c2 s-a făcut prin apelul constructorului de copiere generat de compilator, ca urmare nu se afișează nici un mesaj corespunzător apelului constructorului, în schimb se afișează mesajele corespunzătoare eliminării ambelor obiecte, c1 și c2.

În cazul în care clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază.

Pentru clasa punct din exemplul anterior se include definirea constructorului de copiere:

```
punct(punct &p)
{
    x=p.x;
    y=p.y;
    cout<<"\nConstructor de copiere punct:";
    cout<<"\tx="<<x<<"\ty="<<y;
}
```

La execuția programului, se va afișa:

```
Constructor punct:      x=1.1  y=2.2
Constructor cerc:      raza=10

Constructor de copiere punct: x=1.1  y=2.2

Destructor cerc:      raza=10
Destructor punct:      x=1.1  y=2.2

Destructor cerc:      raza=10
Destructor punct:      x=1.1  y=2.2
```

Se observă afișarea mesajului corespunzător apelului constructorului de copiere pentru clasa punct. Acesta a fost apelat de constructorul de copiere generat de compilator pentru clasa cerc.

În cazul în care se definește constructor de copiere pentru clasa derivată, acestuia îi revine în totalitate sarcina transferării valorilor corespunzătoare membrilor ce aparțin clasei de bază. Deci, pentru clasa cerc se include constructorul de copiere cu prototipul:

cerc(cerc &);

iar antetul este de forma:

cerc::cerc(cerc &c) : punct(c.x, c.y)

```
cerc::cerc(cerc &c) : punct(c.x, c.y)
{
    raza=c.raza;
    cout<<"\nConstructor de copiere cerc: "
    cout<<"\traza="<<raza;
}
```

La execuția programului, se afișează:

```
Constructor punct:      x=1.1  y=2.2
Constructor cerc:      raza=10

Constructor punct:      x=1.1  y=2.2
Constructor de copiere cerc: raza=10

Destructor cerc:      raza=10
Destructor punct:      x=1.1  y=2.2

Destructor cerc:      raza=10
Destructor punct:      x=1.1  y=2.2
```

Se poate observa că, deși este definit constructor de copiere pentru clasa de bază, pentru crearea obiectului punct se apelează constructorul cu parametri, nu cel de copiere.

Lista ce specifică parametrii ce se transmit către obiectul clasei de bază poate lipsi în situația în care pentru clasa de bază este definit constructor implicit sau constructor cu parametri implicați.

## Redefinirea funcțiilor membre

Clasa derivată are acces la toți membrii cu acces protected sau public ai clasei de bază. Astfel, dacă se definește pentru clasa punct o funcție membră de afișare:

```
punct::afisare()
{
    cout<<"\nPunct: x="<<x<<" , y="<<y;
}
```

și se definește funcția main():

```
void main()
{
    cerc c(1.1, 2.2, 10);
    c.afisare();
}
```

rezultatul execuției acestei secvențe este afișarea:

```
Constructor punct:      x=1.1  y=2.2
Constructor cerc:      raza=10

Punct: x=1.1  y=2.2

Destructor cerc:      raza=10
Destructor punct:      x=1.1  y=2.2
```

Funcția de afișare membră a clasei punct este moștenită de clasa cerc, dar, având în vedere faptul că o funcție de afișare pentru clasa cerc ar trebui să ofere mai multă informație, este necesară definirea unei noi funcții de afișare, membră a clasei cerc.

Este permisă supradefinirea funcțiilor membre clasei de bază cu funcții membre ale clasei derivate.

Clasa cerc se completează cu funcția membră: void afisare().

```
void cerc::afisare()
{
    cout<<"\nCerc:";
    cout<<"\ncentru: x="<<x<<"\ty="<<y;
    cout<<"\nraza: "<<raza;
}
```

Apelul funcției :

```
c.afisare();
```

are ca efect afișarea:

```
Cerc:
centru: x=1.1  y=2.2
raza= 10
```

Noile definiții din clasa derivată nu substituie definițiile din clasa de bază, ci se adaugă acestora.

## Compatibilitatea între o clasă derivată și clasa de bază. Conversii de tip

Deoarece clasa derivată moștenește proprietățile clasei de bază, între tipul clasă derivată și tipul clasă de bază se admite o anumită compatibilitate.

Compatibilitatea este valabilă numai pentru clase derivate cu acces public la clasa de bază și numai în sensul de la clasa derivată spre cea de bază, nu și invers.

Compatibilitatea se manifestă sub forma unor conversii implicite de tip:

- dintr-un obiect derivat într-un obiect de bază;
- dintr-un pointer sau referință la un obiect din clasa derivată într-un pointer sau referință la un obiect al clasei de bază.

De exemplu, constructorul de copiere al clasei cerc se poate rescrie sub forma:

```
....
cerc::cerc(cerc & c): punct( c )
{
    raza=c.raza;
    afisare();
}
...
void main()
{
    cerc c1(1.1, 2.2, 10);
    cerc c2(c1);
}
```

Rezultatul execuției programului este:

```
Constructor punct:      x=1.1  y=2.2
Constructor cerc:      raza=10

Constructor de copiere punct: x=1.1  y=2.2
Constructor de copiere cerc:  raza=10

Destructor cerc:      raza=10
Destructor punct:     x=1.1  y=2.2

Destructor cerc:      raza=10
Destructor punct:     x=1.1  y=2.2
```

Prin definiția folosită pentru constructorul de copiere se determină apelul constructorului de copiere al clasei punct în urma conversiei implicite referință cerc -> referință punct.

Având în vedere acceptarea conversiilor implicite, este corectă secvența:

```
|| void test(punct &r_p)
```

```
{ r_p.afisare(); }

void main()
{
    punct p(2.5, 7.1), *p_p;
    cerc c(4.5, 3.2, 15), *p_c;
    p=c;                // conversie cerc->punct la atribuire
    p.afisare();
    punct p1(c);        // conversie cerc->punct la apelul constructorului de copiere
    p1.afisare();
    test(c);            // conversie referință cerc->referință punct la transferul parametrului
                        // prin referință
    p_p=&c;              // conversie pointer cerc -> pointer punct la atribuire
    p_p->afisare();
}
```

Atribuirea `p=c` se face în urma conversiei implicite a obiectului de tip `cerc` `c`, în tipul `punct`.

O atribuire de forma `c=p` nu este permisă, ea fiind semnalată cu mesaj de eroare.

Situații similare apar la transferul parametrilor la apelul funcțiilor prin valoare, referință (vezi exemplul) sau pointer la clasă.

Nu sunt permise conversii inverse. De exemplu atribuirile:

```
p_c=&p;                // conversie pointer punct->pointer cerc
c=p;                  // conversie punct->cerc
cerc &r_c=p;           // conversie referința punct->referința cerc
```

nu sunt permise, ele conducând la erori la compilarea programului.

Dacă într-o aplicație, o conversie de la clasa de bază la clasa derivată este necesară, ea se poate defini prin supradefinirea operatorului de atribuire sau a operatorului cast.

## **Supradefinirea operatorilor în clasele derivate**

Funcțiile operator membre ale clasei de bază sunt moștenite de clasele derivate și pot fi supradefinite în clasele derivate, ca toate celelalte funcții membre.

O excepție o constituie operatorul de atribuire, care se supune următoarelor reguli:

- Dacă operatorul este supradefinit în clasa derivată, funcția `operator=()`

preia sarcina efectuării atribuirilor pentru toți membrii, și pentru cei ai clasei de bază, chiar dacă este supradefinit operatorul în clasa de bază;

- Dacă nu este supradefinită funcția operator=() în clasa derivată, dar este definită în cea de bază, atribuirea pentru membrii clasei de bază se face apelându-se funcția operator=(), pentru ceilalți membri făcându-se atribuirea membru cu membru;
- Dacă nu este supradefinit operatorul de atribuire în nici una din clase, atribuirea se face membru cu membru.

În exemplul următor se declară clasa segment care definește un segment de dreaptă prin coordonatele extremităților sale și clasa derivată din aceasta, dreptunghi care preia segmentul de dreaptă ca și diagonală a sa.

```
#include <iostream.h>

class segment
{
protected:
    int x1, y1, x2, y2;
public:
    segment(int a1=0, int b1=0, int a2=0, int b2=0)
    { cout<<"\nConstructor segment";
      if (a1<a2)
        { x1=a1; x2=a2; }
      else
        { x1=a2; x2=a1; }
      if (b1<b2)
        { y1=b1; y2=b2; }
      else
        { y1=b2; y2=b1; }
    }
    segment operator=(segment s)
    { cout<<"\nApel operator=()pentru segment";
      x1=s.x1, y1=s.y1; x2=s.x2; y2=s.y2;
      return *this;
    }
    segment operator+(segment s)
    { cout<<"\nApel operator+() pentru segment";
      segment aux;
      aux.x1=x1+s.x1;
      aux.y1=y1+s.y1;
      aux.x2=x2+s.x2;
      aux.y2=y2+s.y2;
      return aux;
    }
}
```

```

void afisare()
{
    cout<<"\nx1="<<x1<<" y1="<<y1<<" x2="<<x2<<" y2="<<y2;
}
};

class dreptunghi: public segment
{
public:
    dreptunghi(segment s):segment(s)
    { cout<<"\nConstructor - conversie segment->dreptunghi"; }
    dreptunghi(int a1=0, int b1=0, int a2=0, int b2=0):segment(a1,b1,a2,b2)
    { cout<<"\nConstructor dreptunghi";}
    long aria()
    { return (x2-x1)*(y2-y1);}
    void afisare()
    { segment::afisare();                // se apelează funcția afisare() a clasei de bază
      cout<<"\naria="<<aria();
    }
};

void main()
{
    dreptunghi d1(1, 1, 3, 3), d2(5, 5, 8, 8), d3;
    d3=d1;
    d1.afisare();
    d3=d1+d2;
    d3.afisare();
}

```

În urma execuției programului se afișează:

Constructor segment	// pentru fiecare din cele 3 obiecte dreptunghi se
Constructor dreptunghi	// apelează constructorul segment și apoi constructorul
Constructor segment	// dreptunghi
Constructor dreptunghi	
Constructor segment	
Constructor dreptunghi	
Apel operator=() pentru segment	// Pentru atribuirea d3=d1, în absența supradefinirii
	// operator=() pentru clasa dreptunghi, compilatorul
	// apelează operator=() definit pentru segment
x1=1 y1=1 x2=3 y2=3	
aria=4	
Apel operator+() pentru segment	// Extinderea funcționării operator+() definit
	// pentru segment asupra obiectelor dreptunghi
Constructor segment	// Rezultatul returnat de operator+() e preluat de un
	// obiect temporar segment



Constructor – conversie segment->dreptunghi	// Conversie necesară atribuirii
Apel operator=() pentru segment	// Apel similar celui anterior al funcției
	// operator=()
x1=6 y1=6 x2=11 y2=11	
aria=25	

Expresia  $d3=d1+d2$  se evaluează în următoarele etape:

- se apelează funcția `operator+()` definită pentru clasa `segment`, prin extinderea funcționării ei asupra clasei `dreptunghi`;
- în timpul execuției funcției `operator+()` se creează obiectul de tip `segment` local funcției, `aux`, care va fi returnat de către aceasta;
- se face conversia `segment->dreptunghi` definită prin constructor (este necesară definirea conversiei deoarece este conversie clasă\_de\_bază -> clasă\_derivată, conversie ce nu poate fi implicită), generându-se un obiect temporar `dreptunghi`;
- compilatorul realizează atribuirea prin apelul implicit al funcției `operator=()` definită pentru clasa `segment`.

### Exercițiu:

Să se definească clasa `dreptunghi` pentru referirea unei zone din ecran prin poziția colțurilor stânga-sus și dreapta-jos, cu specificarea culorii de afișare. Preluând această clasă drept clasă de bază, să se definească clasa derivată `dreptunghi_cu_chenar` prin care se specifică caracterul folosit pentru trasarea chenarului și culoarea acestuia. Afișarea se face în modul `text`.

### Crearea unei ierarhii de clase

Clasele derivate pot fi utilizate, la rândul lor drept clase de bază pentru a obține noi clase derivate, creându-se astfel ierarhii de clase. Se pornește de la o clasă relativ simplă. La fiecare derivare se adaugă noi proprietăți obținându-se clase tot mai complexe. Fiecare clasă derivată moștenește proprietățile tuturor claselor utilizate în mod succesiv drept clase de bază.

În exemplul următor se creează o ierarhie clasa poz -> clasa punct -> clasa caracter. Clasa caracter moștenește proprietățile celorlalte două.

```
#include <iostream.h>
#include <conio.h>

class poz // declarația clasei poz
{
protected:
    int x, y;
public:
    poz(int abs=0, int ord=0)
    { cout<<"\nconstructor poz: ";
      x=abs; y=ord;
      cout<<"x="<<x<<"\ty="<<y ;
    }
    ~poz()
    { cout<<"\ndestructor poz:";
      cout<<"x="<<x<<"\ty="<<y ;
    }
    void af()
    {cout<<"\npoz: x="<<x<<" y="<<y;}
    void depl(int dx, int dy)
    {x+=dx; y+=dy; }
};

class punct:public poz // declarația clasei punct, derivată din clasa poz
{
protected:
```

```

int vizibil;                                //vizibil=0-punct invizibil
int culoare;
public:
punct(int abs, int ord, int cul) : poz(abs, ord)
{ cout<<"\nconstructor punct: ";
  vizibil=1;
  culoare=cul;
  cout<<"vizibil="<<vizibil<<"\tculoare="<<cul;
}
~punct()
{ cout<<"\ndestructor punct";
  cout<<"vizibil="<<vizibil<<"\tculoare="<<culoare;
}
void cul(int c)
{ culoare=c; }
void viz(int v)
{ vizibil=v; }
void afisare()
{ if (vizibil)
  { textcolor(culoare);
    gotoxy(x, y);
    cprintf("*");
  }
}
};

class caracter:public punct                // declarația clasei caracter, derivată din clasa
{                                           // punct; ea va moșteni atât membrii clasei punct,
  char c;                                  // cât și ai clasei poz
public:
  caracter(int abs, int ord, int cul, char ch) : punct(abs, ord, cul)
  { cout<<"\nconstructor caracter: ";
    c=ch;
    cout<<"caracter="<<c;
  }
  ~caracter()
  { cout<<"\ndestructor caracter: ";
    cout<<"caracter="<<c;
  }
  void afisare()
  { if (vizibil)
    { textcolor(culoare);
      gotoxy(x, y);
      cprintf("%c", c);
    }
  }
};

```

```
void main()
{
    clrscr();
    poz p1(5, 5);
    p1.af();
    getch();
    punct pct1(7, 7, 7);
    pct1.af();
    pct1.afisare();
    pct1.cul(15);
    pct1.depl(1, 1);
    pct1.afisare();
    getch();
    caracter c1(12, 12, 15, '#');
    c1.afisare();
    getch();
}
```

La execuția programului se afișează:

```
constructor poz:      x=5   y=5           // se creează obiectul p1
poz:   x=5   y=5
constructor poz:      x=7   y=7           // se creează obiectul pct1, apelându-se
                                           // întâi constructorul poz() și apoi punct ()
constructor punct:    vizibil=1 culoare=7
poz: x=7   y=7
    *
    *
constructor poz:      x=12  y=12          // se creează obiectul c1, apelându-se
                                           // întâi constructorul poz(), apoi punct (),
                                           // apoi caracter()
constructor punct:    vizibil=1 culoare=15
constructor caracter: caracter = #
    #
destructor caracter:  caracter = #         // se distruge obiectul c1, apelându-se
destructor punct:     vizibil=1 culoare=15 // întâi destructorul ~poz(), apoi
destructor poz:       x=12   y=12          // ~punct (), apoi ~caracter()

destructor punct:     vizibil=1 culoare=15 // se distruge obiectul pct1, apelându-se
destructor poz:       x=8    y=8           // întâi destructorul ~poz(), apoi ~punct ()

destructor poz:       x=5    y=5           // se distruge obiectul p1, apelându-se
                                           // destructorul ~poz()
```

Clasa caracter este derivată din clasa punct, care la rândul său este derivată din clasa poz. Ea moștenește membrii x și y de la clasa poz, vizibil și culoare de la clasa punct și adaugă membrul caracter. Funcția af() a clasei poz

este moștenită, putând fi apelată pentru un obiect caracter, în schimb funcția afisare() definită în clasa punct este supradefinită în clasa caracter, pentru un obiect caracter fiind apelată funcția definită în clasa caracter.

## Moștenirea multiplă

Conceptul de moștenire permite crearea de clase noi care moștenesc proprietățile mai multor clase de bază. Moștenirea multiplă aduce mai multă flexibilitate în construirea claselor, rezultatul fiind obținerea unor structuri de clase complexe.

Sintaxa folosită pentru declararea unei clase derivate D este:

**class D: specif\_acces clasa\_baza1, specif\_acces clasa\_baza2,... {...}**

În lista claselor de bază, pentru fiecare clasă de bază, se include specificatorul de acces.

Principiile prezentate la derivarea simplă și la crearea ierarhiilor simple de clase sunt valabile și în cazul derivării multiple.

Prin clasa strpoz se vor crea obiecte ce conțin un șir de caractere împreună cu poziția de afișare. Această clasă se obține prin derivarea din clasele pozitie și string, accesul către acestea fiind public.

```
#include <iostream.h>
#include <string.h>
#include <conio.h>

class pozitie                                // declarare clasă pozitie
{
protected :
    int x, y;
public:
    pozitie(int=0, int=0);
    pozitie(pozitie&);
    ~pozitie();
    void afisare();
    void deplasare(int,int);
};

pozitie::pozitie(int abs, int ord)
{
```

```
x=abs; y=ord;
cout<<"\nConstructor - pozitie";
afisare();
}

pozitie::pozitie(pozitie &p)
{
    x=p.x; y=p.y;
    cout<<"\nConstructor copiere-pozitie";
    afisare();
}

pozitie::~~pozitie()
{
    cout<<"\nDestructor - pozitie";
    afisare();
}

void pozitie::afisare()
{
    cout<<"\npozitie: x="<<x<<" y="<<y;
}

void pozitie::deplasare(int dx, int dy)
{
    x+=dx; y+=dy;
}

class string // declarare clasă string
{
protected:
    int ncar; // lungimea sirului
    char *str;
public:
    string(int n=0)
    {
        ncar=n;str=new char[ncar+1];
        str[0]='\0';
        cout<<"Constructor 1- string" ;
        afisare();
    }
    string(char * s)
    {
        ncar=strlen(s); str=new char[ncar+1];
        strcpy(str,s);
        cout<<"\nConstructor 2 - string";
        afisare();
    }
}
```

```

string(string & s)
{
    ncar=s.ncar;
    str=new char[ncar+1];
    strcpy(str, s.str);
    cout<<"\nConstructor de copiere-string";
    afisare();
}
~string()
{ cout<<"\nDestructor- string";
  afisare();
  delete str;
}
void afisare()
{
    cout<<"\nstring:"<<str<<"\n";
}
string operator+(string &s)
{
    string temp(ncar+s.ncar);
    strcat(temp.str, str);
    strcat(temp.str, s.str);
    return temp;
}
};

class strpoz : public pozitie, public string           // declararea clasei strpoz, derivată din
                                                       // pozitie și string cu acces public către
                                                       // acestea
{
    char culoare;
public:
// în antetul constructorului se specifică parametrii preluați de constructorii claselor de bază
    strpoz(int abs, int ord, int n=0, char c='A') : pozitie(abs, ord), string(n)
    {
        culoare=c;
        cout<<"\nConstructor 1 - strpoz";
        afisare();
    }
    strpoz(int abs, int ord, char *s, char c='A'):pozitie(abs, ord), string(s)
    {
        culoare=c;
        cout<<"\nConstructor 2 - strpoz";
        afisare();
    }
    strpoz(strpoz &sp) : pozitie(sp), string(sp)      // parametrul preluat de constructorii
                                                       // claselor de bază este chiar sp pentru
                                                       // care se face conversia implicită către
                                                       // clasa de bază respectivă

```

```

    { culoare=sp.culoare;
      cout<<"\nConstructor copiere - strpoz";
      afisare();
    }
    ~strpoz()
    {
      cout<<"\nDestructor -strpoz";
      afisare();
    }
    void coloreaza(char c)
    { culoare=c; }
    void afisare()
    {
      cout<<"\nstrpoz:"<<str;
      cout<<"\nx="<<x<<" y="<<y;
      cout<<"\nculoare:"<<culoare<<"\n";
    }
    strpoz operator+(strpoz &sp)
    {
      strpoz temp(x+sp.x, y+sp.y, ncar+sp.ncar);
      strcat(temp.str, str);
      strcat(temp.str, sp.str);
      return temp;
    }
};

void main()
{
  strpoz sp1(5, 5, "TEXT");
  strpoz sp2(sp1);
  string s1("aaa"), s2("bbb");
  string s3=s1+s2;
  s3.afisare();
  strpoz sp3=sp1+sp2;
  sp3.afisare();
}

```

Programul afișează:

Constructor – pozitie	// se creează obiectul sp1, apelându-se, în ordine
pozitie: x=5 y=5	// constructorii pozitie(), string(), strpoz()
Constructor 2 - string	
string : TEXT	
Constructor 2 – strpoz	
strpoz: TEXT	
x=5 y=5	



culoare: A	
Constructor copiere – pozitie pozitie: x=5 y=5	// se creează obiectul sp2, copie a obiectului // sp1, apelându-se, în ordine constructorii de copiere
Constructor copiere - string string : TEXT	
Constructor copiere – strpoz strpoz: TEXT	
x=5 y=5 culoare: A	
Constructor 2 – string string: aaa	// se creează obiectul string s1
Constructor 2 – string string: bbb	// se creează obiectul string s2
Constructor 1 – string string:	// se creează obiectul temp local funcției operator+()
Constructor de copiere – string string: aaabbb	// se creează obiectul s3, copie a obiectului returnat de // funcția operator+()
Destructor – string string: aaabbb	// se elimină obiectul temp la ieșirea din funcția // operator+()
string: aaabbb	// apel al funcției afisare() pentru obiectul string s3
Constructor copiere – pozitie pozitie: x=10 y=10	// se creează obiectul temp de tip strpoz , apelându-se, în // ordine constructori pentru clasele pozitie,string, strpoz
Constructor 1 – string string:	
Constructor 1 – strpoz strpoz:	
x=10 y=10 culoare=A	
Constructor copiere – pozitie Pozitie: x=10 y=10	// se creează obiectul sp3, prin copierea obiectului // returnat de funcția operator+()
Constructor copiere - string string : TEXTTEXT	
Constructor copiere - strpoz strpoz : TEXTTEXT	
x=10 y=10 culoare: A	
Destructor - strpoz strpoz : TEXTTEXT	// se elimină obiectul temp la ieșirea din funcția // operator+(), apelându-se destructorii în ordinea:

```
x=10 y=10 //~strpoz(), ~string(), ~pozitie()
culoare: A
Destructor – string
string: TEXTTEXT
Destructor – pozitie
pozitie: x=10 y=10

strpoz:TEXTTEXT // se afișează obiectul sp3
x=10 y=10
culoare:A

Destructor - strpoz // se elimină obiectul sp3 la ieșirea din funcția main(),
strpoz : TEXTTEXT // apelându-se destructorii în ordinea: ~strpoz(),
x=10 y=10 // ~string(), ~pozitie()
culoare: A
Destructor – string
string: TEXTTEXT
Destructor – pozitie
pozitie: x=10 y=10

Destructor – string // se elimină obiectul s3 la ieșirea din funcția main(),
string:aaabbb // apelându-se destructorul ~string()
Destructor – string // se elimină obiectul s2 la ieșirea din funcția main(),
apelându-se // destructorul ~string()
string:bbb

Destructor – string // se elimină obiectul s1 la ieșirea din funcția main(),
string:aaa // apelându-se destructorul ~string()

Destructor – strpoz // se elimină obiectul sp2 la ieșirea din funcția main(),
strpoz:TEXT // apelându-se destructorii în ordinea: ~strpoz(),
x=5 y=5 // ~string(), ~pozitie()
culoare=A
Destructor – string
string:TEXT
Destructor – pozitie
pozitie: x=5 x=5

Destructor – strpoz // se elimină obiectul sp2 la ieșirea din funcția main(),
strpoz:TEXT // apelându-se destructorii în ordinea: ~strpoz(),
x=5 y=5 // ~string(), ~pozitie()
culoare=A
Destructor – string
string:TEXT
Destructor – pozitie
pozitie: x=5 x=5
```

În declarația clasei strpoz se specifică derivarea cu acces public din clasele pozitie și string.

Definiția fiecărui constructor al clasei strpoz specifică transferul datelor către constructorii claselor de bază.

Constructorii și destructorii claselor de bază sunt apelați automat la crearea, respectiv distrugerea obiectelor derivate. Ordinea apelării constructorilor este dată de ordinea din lista claselor de bază din definiția clasei derivate, constructorul clasei derivate fiind apelat ultimul. Destructorii sunt apelați în ordine inversă.

Clasa derivată conține toți membrii claselor de bază, dar îi poate accesa doar pe cei declarați cu acces public sau protected.

Clasele derivate pot supradefini funcții existente în clasele de bază. Membrii și funcțiile omonime pot fi accesate folosind operatorul de rezoluție. De exemplu, pentru clasa strpoz se poate defini funcția membră de afișare astfel:

```
void strpoz::afisare()
{
    cout<<"\nStrpoz:"
    pozitie::afisare();
    string::afisare();
    cout<<"\nculoare:"<<c<<endl;
}
```

iar în funcția main() se poate include linia de program:

```
sp3.pozitie::afisare();
```

### Exercițiu:

Să se definească o clasă derivată “produs” care descrie un produs prin denumire, cod, preț. Se vor folosi o clasă de bază string pentru membrul denumire și o clasă care definește codul prin trei grupe de caractere. Se vor realiza cele două variante de derivare, și anume o ierarhie simplă de clase și, respectiv, derivarea multiplă.

### Clase virtuale

În situația unei derivări multiple, în lista claselor de bază se pot regăsi clase, derivate din aceeași clasă de bază. Problema care apare este faptul că noua clasă obținută va conține membri duplicați. Aceștia pot fi referiți folosindu-se operatorul de rezoluție ::.

De exemplu, se consideră situația:

```
class Baza { protected x ;...};  
class B1: public Baza {...};  
class B2 : public Baza {...};  
class D: public B1, public B2 {...};
```

Clasa D conține doi membri x. Cel care provine din clasa B1 se poate referi sub forma:

Baza::B1::x

iar cel care provine din clasa B2 sub forma:

Baza::B2::x

De obicei această duplicare nu este necesară. Pentru a prelua membrul neduplicat, se declară clasa Baza virtuală în declarațiile claselor B1 și B2.

```
class Baza { protected x ;...};  
class B1: public virtual Baza {...};  
class B2 : public virtual Baza {...};  
class D: public B1, public B2 {...};
```

Rezultatul utilizării cuvântului cheie **virtual** este faptul că în clasa derivată D se va include un singur membru x. Declararea clasei Baza virtuală nu

are efect asupra clase Baza, ci numai asupra claselor derivate din aceasta.

La definirea constructorului clasei derivate este necesar să se precizeze nemijlocit transferul de informație pentru constructorul clasei Baza în vederea creării copiei unice a obiectului Baza. În această situație, definirea constructorului va avea sintaxa:

$$D(...): B1(...), B2(...), Baza(...) \{...\}$$

Într-o ierarhie de clase, constructorul clasei virtuale este întotdeauna apelat primul.

În exemplul următor se definește clasa poz ce descrie poziția unui punct prin coordonatele sale, care va fi folosită ca și clasă de bază pentru clasele cerc și, respectiv pătrat. În final se declară clasa fig care reprezintă o figură formată dintr-un cerc și un pătrat cu centru comun.

```
# include <iostream.h>

class poz                                // se declară clasa poz
{
protected:
    int x, y;
public:
    poz(int abs=0, int ord=0)            // constructor cu parametri impliciti
    {
        cout<<"\nConstructor poz";
        x=abs;
        y=ord;
    }
    void afisare()
    {
        cout<<"\nx="<<x<<" y="<<y;
    }
};

class cerc: public poz                  // se declară clasa cerc derivată din clasa de bază
{
protected:
    float raza;
public:
    cerc(int abs, int ord, float r): poz(abs, ord)    // la definirea constructorului se specifică
                                                        // parametrii preluați de constructorul
                                                        // clasei de bază
    {
        cout<<"\nConstructor cerc";
        raza=r;
    }
}
```

```
void afisare()
{
    cout<<"\nx="<<x<<" y="<<y;
    cout<<"\nraza="<<raza;
}
};

class patrat: public poz
// se declară clasa patrat derivată din clasa
// de bază poz
{
protected:
    float latura;
public:
    patrat(int abs, int ord, int l) : poz(abs, ord) // la definirea constructorului se specifică
// parametrii preluați de constructorul
// clasei de bază
    {
        cout<<"\nConstructor patrat";
        latura=l;
    }
    void afisare()
    {
        cout<<"\nx="<<x<<" y="<<y;
        cout<<"\nlatura="<<latura;
    }
};

class fig : public cerc, public patrat
// se declară clasa fig derivată din cerc și
// patrat
{
public:
// la definirea constructorului se specifică parametrii preluați de constructorii claselor de bază
    fig(int abs, int ord, float dim) : cerc(abs, ord, dim/2.): patrat(abs, ord, dim)
    {
        cout<<"\nConstructor fig";
    }
    void afisare()
    {
        cout<<"\nx="<<cerc::x<<" y="<<cerc::y;
        cout<<"\nx="<<patrat::x<<" y="<<patrat::y;
        cout<<"\nraza="<<raza<<" latura="<<latura;
    }
};

void main()
{
    fig f(5, 10, 25);
    f.afisare();
}
```

La execuția programului se afișează:

```
Constructor poz  
Constructor cerc  
Constructor poz  
Constructor patrat  
Constructor fig  
x=5 y=10  
x=5 y=10  
raza=12.5 latura=25
```

Se observă că, la crearea obiectului `f` se creează întâi un obiect `cerc`, pentru care se apelează inițial constructorul `poz()` și apoi constructorul `cerc()`, apoi un obiect `patrat` pentru care se apelează întâi constructorul `poz()` și apoi constructorul `patrat()`, și, în final se apelează constructorul `fig`. Pentru crearea unui obiect `fig` se apelează de două ori constructorul `poz()`, deci vor exista câte doi membrii `x` și `y`. Pentru a fi distinși în funcțiile membre clasei `fig`, acești membri trebuie referiți prin numele clasei căreia aparțin. O definiție a funcției de afișare sub forma:

```
void fig::afisare()  
{  
    cout<<"\nx="<<x<<" y="<<y;  
    ....}
```

generează un mesaj de eroare, deoarece compilatorul nu știe la care membru `x`, respectiv `y`, se face referire.

Dacă, în momentul declarării claselor `cerc` și `patrat`, clasa de bază este însoțită de specificația virtual, la crearea unui obiect `fig` constructorul `poz` se apelează o singură dată, membrii `x` și `y` fiind unici. Declarațiile claselor `cerc` și `patrat` vor fi:

```
class cerc : virtual public poz  
{...};  
class patrat : virtual public poz  
{...};
```

Funcția `afisare()` a clasei `fig` poate fi definită astfel:

```
void fig::afisare()  
{  
    cout<<"\nx="<<x<<" y="<<y;  
    cout<<"\nraza="<<raza<<" latura="<<latura;  
}
```

fără a genera erori.

La execuția programului se va afișa:

```
Constructor poz  
Constructor cerc  
Constructor patrat  
Constructor fig  
x=5 y=10  
raza=12.5 latura=25
```

Se observă că se apelează o singură dată constructorul clasei poz(), acesta fiind primul constructor apelat.

## Funcții virtuale

Așa cum s-a arătat în lucrarea anterioară, între o clasă derivată și clasa de bază se admite o anumită compatibilitate în sensul conversiei de la clasa derivată spre cea de bază, nu și invers. Conversiile implicite sunt acceptate atât pentru obiecte de tip clasă derivată, cât și pentru referințe sau pointeri ai acestora.

Presupunem că avem o clasă de bază, B și clasele derivate D1, D2, D3, ..., care redefinesc o metodă M. Se pune problema modului în care compilatorul identifică corect care metodă va fi apelată. În situația utilizării operatorului de scop (un apel de forma B::M()) sau apelului cu ajutorul obiectului asupra căruia se aplică metoda (de exemplu o exprimare de forma: D1 d; d.M();), decizia este simplă și este luată în faza de compilare. În acest caz este vorba despre **legătura statică** (în terminologia engleză “**early binding**”). Există însă situații în care un pointer la clasa de bază poate primi pe parcursul execuției programului ca valoare adrese de obiecte de tip clasă derivată sau clasă de bază, ceea ce presupune luarea unei decizii în privința metodei care se apelează chiar în timpul execuției programului. Acest mod de lucru se numește **legătură dinamică** (“**late binding**”). Funcțiile membre pentru care se realizează legătura dinamică se numesc **funcții virtuale** și se declară cu ajutorul cuvântului cheie **virtual**. Redefinirea cu același prototip în întreaga ierarhie de clase a unei funcții declarată virtuală în clasa de bază, se supune legăturii dinamice. Funcțiile virtuale au următoarele proprietăți:

- sunt funcții membre nestatice ale unei clase;
- redefinirea funcției virtuale se face cu respectarea prototipului;
- redefinirea funcției virtuale în clasele derivate nu este obligatorie;
- redefinirea unei funcții virtuale cu schimbarea prototipului, are ca efect



supradefinirea funcției; funcțiilor supradefinite nu li se mai aplică legătura dinamică;

- constructorii nu pot fi funcții virtuale, în schimb destructorii da;
- funcțiile inline nu pot fi virtuale.

Se declară ierarhia simplă de clase poz->punct->carcter (vezi lucrarea nr. 8).

```
#include <iostream.h>

class poz                                // declarare clasă poz
{
protected:                             // se permite accesul claselor derivate către datele
// membre
    int x, y;
public:
    poz(int =0, int=0);
    virtual void afisare();              // funcția afisare() se declară virtuală
    void deplasare(int, int);
};

class punct:public poz                  // declararea clasei derivate punct
{
protected:                             // se permite accesul claselor derivate către datele
// membre
    int vizibil;
    int culoare;
public:
    punct(int=0, int=0, int=1);
    void afisare();                      // se redefinește funcția virtuală afisare()
    void deplasare(int, int);
};

class caracter:public punct             // declararea clasei derivate caracter
{
    char c;
public:
    caracter(int, int, int, char);
    void afisare();                      // se redefinește funcția virtuală afisare()
    void deplasare(int, int);
};

poz::poz(int abs, int ord)
{
    cout<<"\nConstructor pozitie:"<<this;
    x=abs;    y=ord;
```

```
    cout<<" x="<<x<<" y="<<y;
}
void poz::afisare()
{
    cout<<"\nPozitie:"<<this;
    cout<<" x="<<x<<" y="<<y<<endl;
}
void poz::deplasare(int dx, int dy)
{
    x+=dx;    y+=dy;
    cout<<"\nDeplasare poz:";
    afisare();
}

punct::punct(int abs, int ord, int cul):poz(abs, ord)
{
    cout<<"\nConstructor punct:"<<this;
    culoare=cul;    vizibil=0;
    cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil="<<vizibil;
}
void punct::afisare()
{
    cout<<"\nPunct:"<<this;
    cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil="<<vizibil<<endl;
}
void punct::deplasare(int dx, int dy)
{
    x+=dx;    y+=dy;
    cout<<"\nDeplasare punct:";
    afisare();
}

caracter::caracter(int abs, int ord, int cul, char ch):punct(abs, ord, cul)
{
    cout<<"\nConstructor caracter:"<<this;
    c=ch;
    cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil="<<vizibil;
    cout<<" caracter="<<c;
}
void caracter::afisare()
{
    cout<<"\nCaracter:"<<this;
    cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil="<<vizibil;
    cout<<" caracter="<<c<<endl;
}
void caracter::deplasare(int dx, int dy)
{
    x+=dx;    y+=dy;
```

```

        cout<<"\nDeplasare caracter:";
        afisare();
    }

void main()
{
    // se declară obiecte de tip clasă
    poz p1(1, 1);
    punct p2(2, 2, 1);
    caracter c1(3, 3, 3, '*');

    // se declară pointeri la clase cu inițializare, între tipul pointerilor și tipul obiectului existând
    // corespondență
    poz* p_poz=&p1;
    punct * p_punct=&p2;
    caracter * p_car=&c1;

    // apeluri ale funcțiilor membre
    p_poz->afisare();
    p_poz->deplasare(2, 2);
    p_punct->afisare();
    p_punct->deplasare(3, 3);
    p_car->afisare();
    p_car->deplasare(5, 5);

    p_poz=p_punct;           // atribuire admisă, prin conversia implicită punct*->poz*;
    p_poz->afisare();
    p_poz->deplasare(3, 3);

    p_poz=p_car;             // atribuire admisă, prin conversia implicită caracter*->poz*;
    p_poz->afisare();
    p_poz->deplasare(3, 3);
}

```

Programul afișează:

Constructor pozitie: 0xfff0	x=1	y=1		
Constructor pozitie: 0xffe6	x=2	y=2		
Constructor punct: 0xffe6	x=2	y=2	culoare=1	vizibil=0
Constructor pozitie: 0xffd8	x=3	y=3		
Constructor punct: 0xffd8	x=3	y=3	culoare=1	vizibil=0
Constructor caracter:0xffd8	x=3	y=3	culoare=1	vizibil=0 caracter=*
Pozitie: 0xfff0	x=1	y=1		
Deplasare poz:				
Pozitie: 0xfff0	x=1	y=1		

```
Punct: 0xffe6      x=2   y=2   culoare=1   vizibil=0

Deplasare punct:
Punct: 0xffe6      x=3   y=3   culoare=1   vizibil=0

Caracter:0xffd8    x=3   y=3   culoare=1   vizibil=0 caracter=*

Deplasare caracter:
Caracter:0xffd8    x=8   y=8   culoare=1   vizibil=0 caracter=*

// Se poate observa că, deși p_poz este definit ca pointer la poz, el conține adresa unui obiect
// punct, deci, datorită faptului că funcția afisare() este virtuală, se va apela definiția funcției
// corespunzătoare clasei punct
Punct: 0xffe6      x=5   y=5   culoare=1   vizibil=0

// Funcția deplasare() nu este virtuală, deci se va apela definiția funcției corespunzătoare
// clasei poz, dar funcția afisare() apelată din interiorul acesteia este corect apelată
Deplasare poz:
Punct: 0xffe6      x=8   y=8   culoare=1   vizibil=0

// Se poate observa că, deși p_poz este definit ca pointer la poz, el conține adresa unui obiect
// caracter, deci, datorită faptului că funcția afisare() este virtuală, se va apela definiția funcției
// corespunzătoare clasei caracter
Caracter:0xffd8    x=8   y=8   culoare=3   vizibil=0 caracter=*

// Funcția deplasare() nu este virtuală, deci se va apela definiția funcției corespunzătoare
// clasei poz, dar funcția afisare() apelată din interiorul acesteia este corect apelată
Deplasare poz:
Caracter:0xffd8    x=11  y=11  culoare=3   vizibil=0 caracter=*
```

Din exemplul prezentat se observă că apelul funcției afisare() se face corespunzător obiectului declarat, chiar dacă pointerul este de tipul clasei de bază, datorită faptului că funcția a fost declarată virtuală. Funcția deplasare(), ne fiind virtuală, va fi apelată corespunzător tipului pointerului.

Dacă se înlocuiește declarația funcției deplasare() a clasei poz cu declarația:

virtual void deplasare(int, int);

se va observa în urma execuției programului că și apelul acesteia se face corespunzător tipului obiectului și nu tipului pointerului. În această situație, corespunzător secvenței

```
| p_poz=p_punct;
```

```
p_poz->afisare();
p_poz->deplasare(3, 3);

p_poz=p_car;
p_poz->afisare();
p_poz->deplasare(3, 3);
```

din funcția main(), se va afișa:

```
Punct: 0xffe6      x=5    y=5    culoare=1    vizibil=0
Deplasare punct:
Punct: 0xffe6      x=8    y=8    culoare=1    vizibil=0
Caracter:0xffd8    x=8    y=8    culoare=3    vizibil=0  caracter=*
Deplasare caracter:
Caracter:0xffd8    x=11   y=11   culoare=3    vizibil=0  caracter=*
```

În clasele derivate, dacă pentru declararea unei funcții declarată virtuală în clasa de bază se folosește un prototip diferit, funcția va fi supradefinită și se pierde legătura dinamică. Acest lucru este acceptat de compilator, dar se afișează un mesaj de atenționare a faptului că noua declarație ascunde existența funcției virtuale.

De exemplu, în clasa caracter se va înlocui declarația funcției deplasare cu declarația:

void deplasare();

cu definiția:

```
void caracter::deplasare()
{
    x+=1;    y+=1;
    cout<<"\nDeplasare caracter:";
    afisare();
}
```

În acest caz, apelul funcției se va face astfel:

```
p_car->deplasare();           // funcția este fără parametri
```

Pentru secvența:

```
p_poz=p_car;  
p_poz->afisare();  
p_poz->deplasare(3, 3);
```

se va afișa:

```
Caracter:0xffd8      x=8   y=8   culoare=3   vizibil=0  caracter=*  
Deplasare punct:  
Caracter:0xffd8      x=11  y=11  culoare=3   vizibil=0  caracter=*
```

Se poate observa că funcția selectată `deplasare()` apelată, datorită existenței parametrilor din apel, este funcția moștenită de la clasa `punct`.

Apelul :

```
p_poz->deplasare();
```

va fi sancționat cu mesaj de eroare, se cer parametri pentru funcția `deplasare()`, deoarece pointerul la `poz` încearcă să acceseze funcția virtuală.

Nu este obligatorie redefinirea funcțiilor virtuale în clasele derivate, astfel că, dacă eliminăm declarația funcției `deplasare()` din clasa `caracter`, se va apela redefinirea din ultima clasă ierarhic moștenită, deci apelul:

```
p_poz->deplasare(3, 3);
```

va avea ca efect afișarea:

```
Deplasare punct:  
Caracter:0xffd8      x=11  y=11  culoare=3   vizibil=0  caracter=*
```

Având în vedere că folosirea funcțiilor virtuale presupune un consum de memorie suplimentar și operații suplimentare care au ca urmare creșterea timpului de execuție, se recomandă evitarea utilizării nejustificate a funcțiilor virtuale.

### Exercițiu:

Să se redefinească clasele `strpoz`, `pozitie`, `string` declarate în lucrarea nr. 9, utilizând funcții virtuale. Să se urmărească modul de selectare a acestora când sunt apelate prin pointeri de tipuri diferite la clasele definite.